DISS. ETH NO. 28732

Performance Evaluation of Networked Systems

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

SIMON ARNOLD KASSING

Master of Science ETH in Computer Science, ETH Zurich

born on 12.06.1994

citizen of the Netherlands

accepted on the recommendation of

Prof. Dr. Gustavo Alonso (ETH Zurich), examiner
Prof. Dr. Timothy Roscoe (ETH Zurich), co-examiner
Prof. Dr. Laurent Vanbever (ETH Zurich), co-examiner
Prof. Dr. Katerina Argyraki (EPFL), co-examiner

2022

ABSTRACT

Networked systems are tasked with processing large amounts of data and timely responding to requests coming in at high rates. These tasks involve the collaboration and communication between many nodes to achieve high throughput and low latency. Moreover, the system must communicate with any end-user to receive the request and relay the response. The effective and efficient use of the network is of paramount importance to good operation in both cases: within the data center and across the Internet. Networked systems require network services both expressive enough and with sufficient guarantees to enable performant communication of the system over the network. Moreover, it is desirable for its network service to be continuously validated by a monitoring system, to confirm it operates in accordance with its guarantees and satisfies the application's communication needs – even in challenging scenarios such as in the presence of adversaries. Through the development of models and tools, we gain understanding of the performance of networked systems. This enables us to identify bottlenecks and effective improvements, as well as provision these networked systems to deliver sufficiently good performance in a cost-efficient manner, eschewing intolerable performance due to under-provisioning and excessive cost due to over-provisioning.

In this dissertation, we first consider a new network service primitive which permits bounded degradation of delivery and performance in order to speed-up co-located network flows. Second, we provide a novel perspective on performance by considering how to make a networked system along with its monitoring system robust even in the face of an in-network programmable adversary. Third, we develop a simulator for low Earth orbit constellations of satellites which enables convenient performance analysis of such highly dynamic and constantly evolving networked systems. Fourth, we investigate the tradeoff of cost and performance, and build an advisor to cost-efficiently provision a networked system of serverless functions designed to process interactive queries on cold data. With this varied set of contributions, we improve the performance, resilience, analyzability and efficiency of networked systems.

ZUSAMMENFASSUNG

Vernetzte Systeme haben die Aufgabe, große Datenmengen zu verarbeiten und zeitnah auf Anfragen zu reagieren, die mit einer hohen Rate eingehen. Diese Aufgaben benötigen die Zusammenarbeit und Kommunikation zwischen vielen Knoten, um einen hohen Durchsatz und eine geringe Latenz zu ermöglichen. Darüber hinaus muss das System mit jedem Endbenutzer kommunizieren, um die Anfrage zu erhalten und die Antwort weiterzuleiten. Die effektive und effiziente Nutzung des Netzwerks ist in beiden Fällen von größter Bedeutung für einen guten Betrieb: innerhalb des Rechenzentrums und über das Internet. Vernetzte Systeme benötigen Netzwerkdienste, die sowohl ausdrucksstark als auch ausreichend Garantien bieten, um eine performante Kommunikation des Systems über das Netzwerk zu ermöglichen. Darüber hinaus ist es wünschenswert, dass dieser Netzwerkdienst kontinuierlich von einem Überwachungssystem validiert wird, um zu bestätigen, dass er in Übereinstimmung mit seinen Garantien arbeitet und die Kommunikationsanforderungen der Anwendung erfüllt – selbst in herausfordernden Szenarien wie in der Gegenwart von Widersachern. Durch die Entwicklung von Modellen und Werkzeugen gewinnen wir Verständnis für die Leistungsfähigkeit vernetzter Systeme. Dies ermöglicht es uns, Engpässe und effektive Verbesserungen zu identifizieren und diese vernetzten Systeme so bereitzustellen, dass sie auf kosteneffiziente Weise eine ausreichend gute Leistung erbringen und eine nicht tolerierbare Leistung aufgrund von Unterversorgung und übermäßige Kosten aufgrund von Überversorgung vermeiden.

In dieser Dissertation betrachten wir zuerst ein neues Netzwerkdienst-Grundelement, das eine begrenzte Verschlechterung der Bereitstellung und Leistung zulässt, um den Netzwerkfluss in der gleichen Umgebung zu beschleunigen. Zweitens bieten wir eine neue Perspektive auf die Leistung, indem wir überlegen, wie ein vernetztes System zusammen mit seinem Überwachungssystem selbst gegenüber einem programmierbaren Widersacher im Netzwerk robust gemacht werden kann. Drittens entwickeln wir einen Simulator für erdnahe Satellitenkonstellationen, welcher eine bequeme Leistungsanalyse solch hochdynamischer und sich ständig weiterentwickelnder vernetzter Systeme ermöglicht. Viertens untersuchen wir den Kompromiss zwischen Kosten und

Leistung und erstellen einen Ratgeber zur kosteneffizienten Bereitstellung eines vernetzten Systems serverloser Funktionen, das zur Verarbeitung interaktiver Abfragen auf kalten Daten entwickelt wurde. Mit den vielseitigen Beiträgen dieser Dissertation verbessern wir die Leistung, Belastbarkeit, Analysierbarkeit und Effizienz vernetzter Systeme.

ACKNOWLEDGMENTS

First and foremost, I thank my advisor Prof. Gustavo Alonso for his guidance, feedback, advice and insight. I would like to thank Prof. Ankit Singla for advising me in the first part of my doctoral studies. I thank my co-advisor Prof. Timothy Roscoe and committee members Prof. Laurent Vanbever and Prof. Katerina Argyraki for providing insightful feedback.

I was fortunate to have been part of the Systems Group, a wonderful collection of people fostering research and encouraging collaboration and interdisciplinary feedback. I want to thank my colleagues and friends there throughout the years: Vojislav, Debopam, Ingo, Melissa, Elham, Catalina, Max, Dimitris, Fabio, Zhenhao, Michal, Dario, Dan, Wenqi, Runbin, Abishek, Monica, Ghislain, Renato, Lefteris, Michal, Lazar, Kaan, Andrea, Roni, Lukas, Cedric, Bojan, Johannes, Luka, Nadia, Jena, Simonetta, and many others. I want to thank Hussain, Hanjing, Ege and Fizza, with whom I had the privilege to work with during the completion of their respective project or thesis in the Systems Group. I want to thank Eitan Zahavi, Alex Shpiner and Chaim Hoch for the support and kindness during my internship at Mellanox. I am grateful to the people I was fortunate to collaborate with: Prof. Laurent Vanbever, Prof. Ce Zhang, Prof. Michael Schapira, Asaf Valadarsky, and many others.

I want to thank my friends, my family, my brother Ruud, and my parents Regie and Patrick for all the support throughout my many years of study. Finally, I thank my partner Merve for her continuous support: providing a listening ear and cheering me up when I needed it most.

CONTENTS

1 Introduction **1**

1.1 Network service primitives . . . 2

1.1.1 Enhanced performance guarantees . . . 2

1.1.2 Flexible guarantees rather than strict . . . 3

1.2 Networked systems monitoring . . . 4

1.2.1 Network monitoring at scale . . . 5

1.2.2 Monitoring robust to adversaries . . . 7

1.3 The right tools for performance analysis . . . 9

1.3.1 Low Earth orbit (LEO) satellite network simulation . . . 9

1.3.2 Provisioning for serverless query processing . . . 10

1.4 Contributions and structure . . . 11

1.5 Publications and preprints . . . 13

2 New primitives for bounded degradation in network service **15**

2.1 Why new networking primitives? . . . 17

2.1.1 Guaranteed partial delivery . . . 17

2.1.2 Bounded deprioritization . . . 18

2.2 Abstract algorithm design . . . 20

2.2.1 Budgeting algorithm for bounded degradation . . . 20

2.2.2 Max-min guarantee in multi-hop . . . 23

2.3 Practical algorithm design . . . 24

2.3.1 Probing mechanism . . . 24

2.3.2 Algorithm . . . 26

2.3.3 Probing consequences and limitations . . . 27

2.4 Implementation . . . 29

2.5 Evaluation . . . 30

2.5.1 Experimental setup . . . 31

2.5.2 Prioritization scheme comparison . . . 31

2.5.3 Decision of partial delivery 34
2.5.4 Challenge of probing 36
2.5.5 Utility of increased flexibility 41
2.6 Discussion & future work 48
2.7 Related work 49
2.8 Summary 51

3 Characterizing and mitigating programmable in-network adversaries 53
3.1 Modeling attackers and operators 55
3.1.1 Attacker model 55
3.1.2 Operator model 56
3.2 Transport layer targeting 59
3.3 Application-aware targeting 62
3.4 Network layer and monitoring system as target 67
3.5 Mitigation 68
3.5.1 Impede classification of key packets 68
3.5.2 Application-aware network monitoring 69
3.6 Related work 71
3.7 Summary 72

4 Analysis and simulation tools for dynamic LEO networks 73
4.1 Background and motivation 75
4.2 Architectural design 76
4.2.1 State generator 77
4.2.2 Packet-level simulator 77
4.2.3 Analysis and visualization 78
4.3 Experimental setup 78
4.4 Utility evaluation 79
4.4.1 Scalability 79
4.4.2 Packet-level interaction 81
4.4.3 General dynamics analysis 84
4.4.4 Visualization 86
4.5 Summary 86

5 Resource allocation in serverless query processing 89
5.1 Why serverless data processing? . . . 90
5.2 Provisioning and configuration . . . 91
5.3 Estimation challenges . . . 92
5.3.1 Data-processing-related limitations . . . 92
5.3.2 Serverless platforms related limitations . . . 93
5.4 General estimation model . . . 95
5.4.1 Start-up . . . 95
5.4.2 Base overhead . . . 96
5.4.3 Process, compress, and network rates . . . 96
5.4.4 Input . . . 97
5.4.5 Exchange(s) . . . 97
5.4.6 Output . . . 97
5.4.7 Postprocessing . . . 98
5.4.8 Requests costs . . . 98
5.4.9 Overall Completion Time and Cost . . . 98
5.4.10 Model analysis . . . 99
5.5 Advisory tool and its evaluation methodology . . . 100
5.5.1 The Pareto knee . . . 100
5.5.2 Advisory tool and evaluation . . . 101
5.6 Applying the model: Lambada . . . 102
5.6.1 Description . . . 103
5.6.2 Instrumentation . . . 103
5.6.3 Applied model . . . 104
5.6.4 Applied model analysis . . . 108
5.7 Into practice . . . 108
5.7.1 Methodology . . . 108
5.7.2 Scan workloads . . . 109
5.7.3 Exchange workloads . . . 111
5.8 Large benchmark . . . 113
5.8.1 Dataset generation . . . 114
5.8.2 Query modelling . . . 114

5.8.3 Experimental setup 115
5.8.4 Results 115
5.9 Related work 120
5.10 Summary 121

6 Conclusion **123**
6.1 Summary 123
6.2 Research outlook 125
6.2.1 Improving and expanding bounded degradation primitives . . . 126
6.2.2 Cross-layer system performance tracing 126
6.2.3 Enhancing LEO network simulation further 127
6.2.4 Enhancing serverless query processing system modeling 128
6.2.5 General outlook 130

Bibliography **131**

1 INTRODUCTION

Recent decades have witnessed a surge in demand for systems capable of handling workloads at scale while achieving high quality of service. These systems are expected to support high request rates to serve its many users while achieving low latency response times, ranging from seconds (*e.g.,* for interactive data analysis queries [147, 158]) to tens or hundreds of milliseconds (*e.g.,* serving web requests [4, 52, 151]) depending on its use case. The processing of a request often requires the collaboration of many compute and storage nodes to go over large amounts of data [130, 174] or to collect the many components its response consists of [52, 96, 151]. Moreover, the system must communicate with the end-user to receive the request and relay the response. The effective and efficient use of the network is of paramount importance to good operation in both cases (within the data center *and* across the Internet). Three aspects are of particular importance to enable good operation. First, we must expose expressive network services to the systems which enable them to communicate over the network with performance guarantees, and in turn we must design systems which properly make use of this interface with the guarantees in mind. Second, we need to monitor deployed systems and their networks to validate that both the network quality of service is being upheld, and that this service is sufficient for the application to meet its own performance objectives. Third, we must understand the most important characteristics of these systems, such that we can provision and further improve performance. The development of analysis and modeling tools is a key enabler in this respect. We describe the background of these topics and the research questions we pose in the upcoming sections: network service primitives (§1.1), networked system monitoring (§1.2), and the analysis tools vital to understanding the performance of networked systems (§1.3). An overview of the thesis components and contributions is depicted in Fig. 1.1. We conclude this introductory chapter with a description of the structure of this thesis and

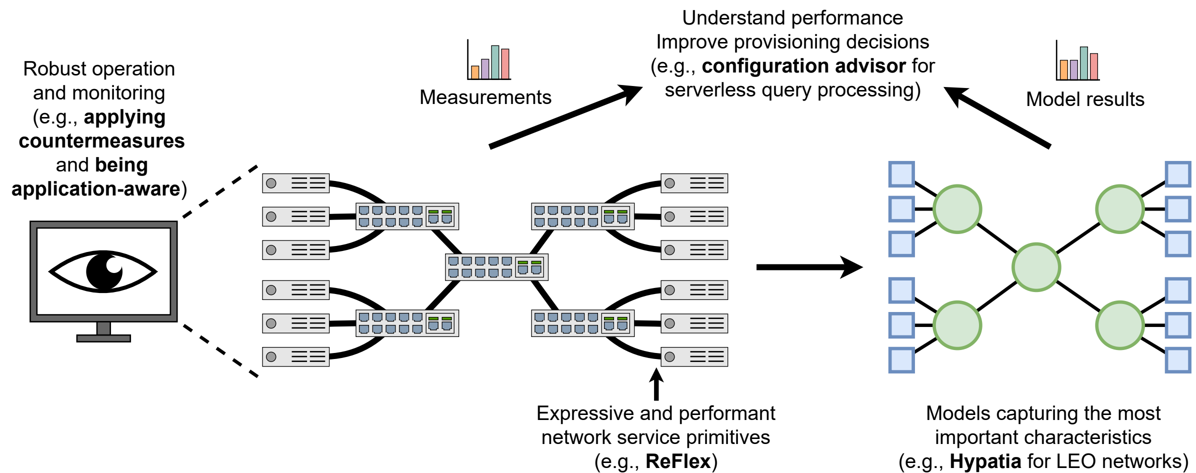


Figure 1.1: Thesis overview.

its contributions (§1.4), and provide the list of related publications and preprints (§1.5).

1.1 Network service primitives

Applications make use of the network service primitives offered to it, and depend on the guarantees associated to it such as reliability and performance through the use of transport protocols. The default available options of either full reliability with fair sharing (TCP) or no reliability (UDP) are not expressive enough to meet the demands of applications, or, due to lack of middle ground, prevent better solutions, which similarly holds for fixed prioritization. There is as such a demand for new network primitives which provide better, more expressive service.

1.1.1 Enhanced performance guarantees

Research has recognized this need for network service with enhanced or specialized performance. Foremost, there has been considerable work in providing data center tenants with network performance guarantees via bandwidth reservation across the network topology [25, 169, 199]. In addition, novel transport protocols have been proposed, both simply incorporating congestion logic different to TCP (*e.g.*, PCC [54]) as well as some requiring certain (minimal) network support. For example, Data Center TCP (DCTCP) [6, 27] makes use router ECN markings to reduce bufferbloat, thus providing improved burst tolerance and lower latency for short flows. Furthermore,

it has been proposed to extend the network service interface with the provision of application knowledge. Several works make use of in-advance knowledge of flow sizes, an assumption which recent work has shown to be *"not necessarily prohibitive"* [194]. pFabric [8] optimizes for average flow completion time, by assigning higher priority to flows with lower remaining flow size combined with switches supporting a custom priority queue. FastPass [159] makes use of a network-aware central scheduler in which slots are allocated along the network path to achieve near-zero queueing. pHost [61] obviates the need of a central scheduler like FastPass, but instead delegates the task of allocation slots to the receivers. Other works have advocated for the application to specify deadlines, and propose a variety of mechanisms for these deadlines to be met. D^3 [203] signals the router along the path of its desired rate, which changes over its lifetime based on the remaining flow size and the approaching deadline – thus requiring custom switch hardware to support this operation. D^2TCP [195] instead increases its congestion window scaling as the deadline approaches to achieve a higher rate by competing more with other flows, thus obviating the need for custom hardware. Sincronia [2] does not optimize for individual flows, instead optimizing for collection of flows of a particular job ("coflows") to finish. These prior works show that there has been considerable work on improved competition of flows by achieving faster and fairer convergence, reducing queueing delay, and satisfying explicit strict guarantees based on application knowledge such as deadlines.

1.1.2 Flexible guarantees rather than strict

In particular, we argue there is opportunity in *flexible* network service models. These models behave according to the principle in which expressed flexibility can be used to accelerate applications which have less or none. In their working, the governing mechanism should abide by the flexibility bounds indicated by applications. Although there are various other degrees of flexibility (*e.g.,* in-order delivery), we believe there is especially promise in exploring the (a) *bounded* partial delivery of data, and (b) *bounded* deterioration of performance. Of the former, there has been work in the machine learning community [210] which has found that network loss which is explicitly bounded to an upper limit can have little to no impact on the performance (*e.g.,* convergence rate) of certain machine learning training tasks. With machine learning becoming an increasingly important cloud workload [67], this is especially promising. Of the latter degree of

flexibility, bounded deterioration, viable use cases are background activity such as backups and administrative tasks, or asynchronous machine learning. It is important to note that for both degrees of flexibility, only partial degradation is permitted: too late completion due to continuously being deprioritized or insufficient amount of delivery is not acceptable to the aforementioned applications.

With the knowledge of the flexibility of flows, the network service can degrade the quality of service of flexible flows in order to accelerate regular or less flexible network flows. Ideally, the presence of these flexible flows should be an absolute boon: acceleration of regular traffic with tolerable impact (within bounds) on the applications which have expressed additional flexibility. Unfortunately, there are no network primitives which provide this service. The default available network services of TCP (fully reliable) and UDP (no reliability) do not provide the ability to express partial delivery bounds. Similarly, network prioritization schemes or scheduling mechanisms do not provide the notion of bounded deprioritization. The concept of applications with more flexible network requirements ceding to more demanding applications has been partially explored. LEDBAT [117] makes use of delay measurements to enforce it does not cause congestion in the network and makes use of a congestion control procedure less aggressive than TCP. The concept of loss flexibility similarly has been explored by Approximate Transport Protocol (ATP) [129]. Under-explored however is the co-design of both loss and deprioritization guarantees, and when data should be considered lost. We formulate the following two research questions:

RQ1: What would be appropriate network service primitives to expose to applications such they can specify partial delivery and bounded degradation?

RQ2: How can partial delivery and bounded degradation primitives be implemented in a data center network?

1.2 Networked systems monitoring

Contemporary networked systems are at a scale exceeding the manual monitoring capability of human operators. Networked systems can comprise hundreds of servers, connected through networks consisting of hundreds to thousands of switches, and an

order of magnitude more interfaces and cables [182]. At this scale, in order to continue to achieve high consistent performance, automatic monitoring systems are employed to oversee networked systems and their network.

1.2.1 Network monitoring at scale

The devices that these networks comprise of, *i.e.,* switches, transceivers and cables, are not infallible: failures at that scale will occur. Moreover, in practice there is no explicit dichotomy of working or not working: there is also partial failure. This category of failures is referred to as *gray failures* [90]. For instance, hardware failures leading to packet corruptions (*e.g.,* connector contamination, bent or damaged cables [220]), linecard faults [168], or software bugs or wrong routing configuration leading to black holes [115].

Gray failures are notoriously difficult to detect, even in the scenario of a data center which is in full control of a single party. This difficulty stems from a variety of factors. First, data center networks encompass a large number of devices, with a bisection bandwidth of multiple terabyte/s supporting thousands of hosts [182]. Second, isolating and taking devices offline for inspection in production data center networks could temporarily impact service [220]. Third, due to the nature of gray failure, pinpointing the cause might be difficult. Gray failures can happen partially (*i.e.,* affect a small fraction of traffic) and possibly only in under certain conditions such as in particular time windows [168]. Fourth, signs of failures such as packet loss, reduced bandwidth or increased latency are not unambiguous. These indicators can similarly be caused by transient congestion due to variance of traffic [168]. Detection of (gray) failures rests on the ability to robustly statistically model what is normal behavior and what is abnormal behavior of the network on fine enough time granularity.

In broad terms, network monitoring systems provide insight by having its resident devices (*e.g.,* end-hosts [23,76,168], switches [49,126,156,219,220]) log (parts of) network traffic or aggregated statistics, which are reported to a central collector. The central collector is responsible for analyzing the incoming data, detecting issues, and potentially tracking down their causes. Subsequently, the system might take automatic actions to fix or temporarily mitigate them (*e.g.,* by disabling [220]), and a human operator is notified with an incident report to facilitate further action (*e.g.,* replacement [220]). It is additionally possible to have an active component in the monitoring system, in which

specific traffic can be generated or tracked for debugging purposes (*e.g.*, by marking traffic [219]).

The most basic network monitoring solutions are sFlow and NetFlow, both of which make use of sampled packets. sFlow [156] directly mirrors samples packets to the central collector, which provides general statistics such as the transport protocol distribution. NetFlow [49] aggregates sampled packets on a per-flow basis, for which it maintains a flow record containing information such as the 5-tuple and the number of observed bytes. These flow records are sent on a regular basis to the central collector. Due to their sampling approach, both sFlow and NetFlow are unable to track packets across the network, as well as might miss important packets at the high sampling rates they operate at [160]. FlowRadar [126] has looked into expansion of NetFlow to monitor every packet such that it also accounts for short small flows, and has shown this to be effective for black hole, switch fault and loop detection [126].

Active monitoring systems such as Pingmesh [76] and NetBouncer [189] actively send out a proportionally small amount of probes into the network. The challenge of these systems is that probes might receive special treatment, or problems might only arise at higher volumes of traffic infeasible for probe traffic without being detrimental to the real traffic. There thus is significant advantage to performing monitoring explicitly on the real traffic. EverFlow [219] enables the matching of every packet, and mirrors packets that it filters as important including TCP control packets, protocol packets, and packets marked with a probe bit to enable manual debugging. Upon packet drop, EverFlow actively injects probe packets over the hop it disappeared on to identify whether the dropping is persistent. It is thus limited to detect gray failure which affected the filtered packets, and might miss problems affecting other parts of the traffic.

Microsoft's 007 [23] is an end-host based approach. Upon observing a retransmission, a path discovery agent at the end-host is activated to perform a traceroute of the TCP connection which encountered the retransmission. The system performs voting on the paths normalized by expected load, and identifies links that are outliers as problematic. Facebook's passive realtime monitoring system [168] (FB-mon) identifies paths by having the aggregation or core switch in their topology mark traversing packets. For each flow, FB-mon gathers TCP state information such as cwnd, ssthresh, and retransmission, and uses this to form equivalent groups based on their paths to identify whether flows going over a particular link experience statistically significant worse performance than another.

To summarize, there is a wide variety of monitoring solutions, which vary on dimensions

such as the mechanism to decide which passing packets (or all) to inspect, and how they are mapped and interpreted to (statistically) infer network performance.

1.2.2 Monitoring robust to adversaries

Our computing ecosystem relies heavily on highly available and performant networks. This critical role has meant that networked systems have long been a high-value target for malicious actors, with high-profile attacks reported with alarming frequency and increasing severity. While the most common attacks target end systems or services like DNS, increasingly, networking devices themselves are coming under attack. The motivation is transparent: compromising network routers and switches provides visibility across large swathes of individual end devices.

Several high-profile attacks on network infrastructure have come to light in recent times, using diverse attack vectors. The attacks tend to affect thousands of switches deployed commonly at ISPs and enterprises [103, 148, 214], are found in certain cases to be perpetrated by state actors [114, 213], and can be focused on particular infrastructure such as data centers [124]. Vulnerabilities in network devices are reportedly frequently; see for example, Cisco's weekly advisories [44], which have disclosed several critical bugs, including in their data center network management software [47], and in software in a popular line of switches [46]. Whitebox devices running Linux variants have also been found to be potentially vulnerable [161].

In summary, while perimeter defense and hardening devices are useful and necessary measures, operators should also examine the question of "What if an attacker does gain control devices in my network?" It is clear that industry leaders are actively considering this possibility even in facilities with best-in-class security:

> *"... it's not that a lot of people do encryption in the data center yet, but that's changing. We're doing that, among other reasons, because **if a malicious actor can break into one switch**, they can snoop on a lot of traffic ..."*
>
> — Jitu Padhye, Microsoft (HotNets-XVI Dialogue, 2017) [146]

While attacks on network devices are not new, they have typically focused on either espionage or disruption. We explore a new type of threat on the horizon, posed by an attacker who has obtained control over programmable switches in a target network:

long-term deterioration in the network's service that is hard to diagnose. While it should be obvious that an attacker controlling powerful network devices can easily degrade performance to the point of denial-of-service, the distinction from well known past attacks is the ability to operate while preventing diagnosis. Preventing or encumbering diagnosis is key to the attack's longevity; if the network operator could easily identify compromised devices, they would be removed or patched. We shall refer to such attacks, conducted with the objective of application performance degradation while preventing diagnosis by making use of programmability, as UNDERRADAR attacks.

RQ3: What are the characteristics of UNDERRADAR attacks?

RQ4: How resilient are networked systems to UNDERRADAR attacks, and what steps can be taken in their design to even further improve their resilience?

Network operators can deploy a variety of monitoring systems that aid in identifying anomalous network behavior and pinpointing faulty devices as is described in §1.2.1. The easiest problems to find are ones involving the total failure of devices, while some of the most difficult to identify, reproduce, and resolve, involve "gray" failures, whereby devices continue to operate, but provide deteriorated functionality. Recent research, particularly in the data center context, has developed increasingly sophisticated methods to detect both total and partial failures, *e.g.,* Microsoft's 007 [23], Everflow [219], and Pingmesh [76], and the passive realtime monitoring system FB-mon tested in a Facebook data center [168]. These systems are a natural adversary for an UNDERRADAR attacker, whose goal is to cause the maximum degradation for applications using the network, while avoiding diagnosis using any deployed monitoring systems. A successful UNDERRADAR attack would result in long-term poor performance, and necessitate time-consuming manual intervention and analysis to eventually fix. It is thus of utmost importance to characterize UNDERRADAR attacks, and devise mitigations to detect and render them ineffective.

RQ5: How effective are existing network monitoring systems in detecting UNDERRADAR attacks, and how can their detection capability be improved further?

1.3 The right tools for performance analysis

The improvement of networked systems hinges on researchers having access to the right tools. These tools enable the evaluation and optimization of existing approaches, the identification of shortcomings, and the discovery of manners to improve them. The level of abstraction of a tool depends on its intended use case and the level of fidelity required to adequately model the key characteristics that impact performance: for instance, it can range from direct calculations to simulation [152], emulation [207], and testbed [43]. Besides their modeling capability, there are important user-oriented constraints, namely *usability* and *scalability*. Firstly, researchers should be able to quickly understand the underlying model and how they can implement their own research methods for evaluation and comparison to others. Secondly, the tools must be able to scale to sufficient system size and continue to return in reasonable time. In this section, we describe two areas which could benefit from improved tools, namely low Earth orbit satellite networks (§1.3.1) and provisioning for serverless query processing (§1.3.2).

1.3.1 Low Earth orbit (LEO) satellite network simulation

Fueled by technological advancements, recent years have witnessed a rising interest in LEO satellite networks. This had led to the first large scale deployments, such as the Starlink constellation by SpaceX [188], the Kuiper constellation by Amazon [119], and the constellation by Telesat [190]. Amongst them, they have already deployed hundreds of satellites and many more are scheduled to be launched in the foreseeable future [74, 132]. LEO satellite networks are poised to supplement or be an alternative to the terrestrial cable-oriented Internet [79]. Unlike their geo-synchronous counterparts, LEO satellites operate at a significantly low enough altitude to provide both low latency and high throughput communication to ground stations on Earth [28, 186]. Moreover, this next generation of satellites are not solely intended to act as relay between two ground stations, but are also set to form a true network in space by communicating with one another through inter-satellite laser links [30, 79, 116].

To fulfill their full potential, LEO satellite networks come with a whole host of engineering challenges, some of which are well-understood whereas others are novel. Unlike cabled networks, the satellite nodes (*i.e.*, the "network switches") are constantly moving at high velocity in their orbit [30]. Over time, their visibility as well as their distance

relative to one another will differ, and the ground stations that connect to them changes. As such, the resulting network is highly dynamic, with varying link properties such as latency and bandwidth, and topological changes of links coming in and out of existence [30]. Faced with this emergent field, network researchers require tools to explore additional questions including topology design, routing and congestion control [28]. Foremost, they want to evaluate to what extent existing methods work, identify incongruity, and develop modifications or new methods to address these challenges. This is additionally even more important as these networks are being deployed now, and as such enhanced modeling capability could enable researchers to steer their deployment to even greater performance heights.

RQ6: Can we build a usable and scalable simulator for low Earth orbit satellite networks which incorporates their highly dynamic topological nature causing varying link properties and changes?

1.3.2 Provisioning for serverless query processing

Serverless computing, now a feature offered by all major cloud providers [15, 69, 139], has been touted as a major step forward in terms of more efficient computing [37, 176]. In spite of the limitations of serverless platforms [85, 88, 200], recent results indicate that analytical query processing over serverless is feasible but that it has a very narrow window where it remains cost effective. The work on Starling [158] and Lambada [147] show that, in current commercial offerings, the maximum throughput is a few tens of queries per hour. Beyond that, it is better to use a fixed infrastructure as it costs less per computation time unit. Nevertheless, these systems show that query processing over serverless can be competitive for interactive, occasional use. For instance, it can be an useful extension to data lakes such as Databrick's Lakehouse [51], Microsoft Azure's Data Lake [138], or Amazon's Redshift [10]. Over large swathes of the data in these large repositories, analytical queries are run only infrequently resulting into what is referred to as "cold" data. Loading such data into traditional databases for processing is costly and impractical. Using a data lake infrastructure is more economical but still incurs too high costs for only occasional use. This is where serverless query processing can be very useful. Compared with using virtual machines, serverless functions have much lower latency to start, are more scalable, and are charged at a smaller time granularity (*e.g.*,

1 ms [14]).

Provisioning resources in the cloud for query processing is not easy [125]. For serverless, the narrow effective window and the limited choices given to the user make the task even more difficult [200]. However, for occasional use over the cold data, such provisioning is critical to keep the system cost-effective. The two obvious extremes yield little practical value to the user: the cheapest choice (small number of the smallest functions) finishes too slow; the fastest (a large number of the largest functions) is too expensive. It is thus desirable to have advisory tool that can recommend suitable configurations for serverless query processing striking a good balance between completion time and financial cost.

RQ7: Can we roughly estimate the performance and cost of serverless query processing systems, and can we use this rough estimation to provision them striking a good balance between completion time and financial cost?

1.4 Contributions and structure

We describe below the overall structure of this thesis. For each chapter, we outline the contributions made therein.

In **chapter 2**, we look at a novel way to make the transport layer level more robust and to satisfy heterogeneous workloads on the same networked system by considering a new flexible network primitive.

- We identify two ways — partial delivery and bounded deprioritization — in which a growing category of data center workloads can allow relaxations of traditional network service expectations. Our approach, REFLEX, adds simple extensions to the application-network interface such that the application can make the network aware of this flexibility. We design a budgeting algorithm to provide guarantees relative to their fair share, which is measured via probing. The requirement of budgeting and probing limits the algorithm's applicability to large flexible flows.
- We evaluate our algorithm with large flexible flows and for three workloads of regular flows of small size, large size and a distribution of sizes. Across the workloads, our algorithm achieves less speed-up of regular flows than fixed prioritization,

especially for the small flows workload. Our algorithm provides better guarantees in the workload with large regular flows. However, it provides not much better or even slightly worse guarantees for the other two workloads.

- We find that the ability to enforce guarantees is influenced by flow fair share interdependence, measurement inaccuracies and dependency on convergence. We observe that priority changes to probe or to deprioritize causes queue shifts which deteriorate guarantees and limit possible speed-up, especially of small flows. We find that mechanisms to both prioritize traffic and track guarantees should be as non-disruptive as possible.

In **chapter 3**, we identify the characteristics of attacks orchestrated by programmable network devices to covertly deteriorate network performance. We explore methods to detect, mitigate and prevent such UNDERRADAR attacks.

- We bring awareness to the notion of UNDERRADAR attacks, which attempt to cause application performance degradation while remaining hard to diagnose through the use of programmability.
- We show that particularly impeding identification of key packets and flows is important, as UNDERRADAR attacks can specifically target transport-layer protocols as well as use knowledge of application structure, such as that of large "coflows", to degrade application performance with fewer interactions. Techniques such as encryption of payload and header, as well as obfuscation of timing and size, we posit to be effective countermeasures in this aspect albeit could limit other network functionality.
- We find that network monitoring systems which continuously monitor real traffic are most effective at detecting such attacks. They should be finely tuned to not only detect distribution shifts which indicate network failure but also outliers that could have been the result of targeted interference.
- We highlight the benefits of making network monitoring systems application-aware, such that degradation in application performance can directly be diagnosed whether it is caused by disproportionally degraded network service.

In **chapter 4**, we describe low-earth orbit (LEO) satellite networks, the research demand to model their highly dynamic characteristics, and the implementation of a usable, scalable and expressive simulator to address this need.

- We develop the simulation framework HYPATIA which models the highly dynamic topological nature of LEO networks, and provide insight into its scalability.
- We show through packet-level experiments the effect LEO network dynamics such as path and base latency changes on network operation, which shows HYPATIA's usability for areas such as congestion control, routing and traffic engineering.

In **chapter 5**, we examine serverless query processing systems, which excel at cost efficiently and in a timely manner answering infrequent analytical queries on large quantities of cold data. We investigate how to provision these systems in a cost-efficient and performant manner.

- We develop a parameterized model which accounts for the key characteristics of serverless query processing (namely, start-up, network, processing, and overhead) which enables the rough estimation of performance and cost;
- We build an advisory model which makes use of these rough estimations to recommend a configuration which achieves a good balance (close to the "Pareto knee") between completion time and financial cost;
- In our evaluation using TPC-H, we show that the model chooses a configuration 21 out of 24 times with less than 15% distance of the comparison metric.

Finally, in **chapter 6**, we conclude the thesis by going over its key takeaways and describe the possible directions for future work. We find there are many more opportunities to further improve networked systems and their monitoring, with better performance at reduced cost in the prospect, and their performance evaluation, with increased ease of insight, scalability and usability.

1.5 Publications and preprints

This dissertation incorporates parts of works which have already been published, which are as follows (in order of appearance in this thesis):

- Simon Kassing, Vojislav Dukic, Ce Zhang, and Ankit Singla. *New primitives for bounded degradation in network service.* arXiv:2208.08429, 2022. [107]

- Simon Kassing, Hussain Abbas, Laurent Vanbever, and Ankit Singla. *Order P4-66: Characterizing and mitigating surreptitious programmable network device exploitation* arXiv:2103.16437, 2021. [105]
- Simon Kassing*, Debopam Bhattacherjee*, André Baptista Águas, Jens Eirik Saethre, and Ankit Singla. *Exploring the "Internet from space" with Hypatia.* ACM IMC, 2020. [106]
- Simon Kassing, Ingo Müller, and Gustavo Alonso. *Resource Allocation in Serverless Query Processing.* arXiv:2208.09519, 2022. [108]

Additionally, during my doctoral studies I was involved in several other works which are not directly covered in this thesis:

- Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. *Beyond fat-trees without antennae, mirrors, and disco-balls.* ACM SIGCOMM, 2017. [110]
- Chen Yu, Hanlin Tang, Cedric Renggli, Simon Kassing, Ankit Singla, Dan Alistarh, Ce Zhang, and Ji Liu. *Distributed learning over unreliable networks.* ICML, 2019. [210]
- Debopam Bhattacherjee*, Simon Kassing*, Melissa Licciardello, and Ankit Singla. *In-orbit computing: An outlandish thought experiment?* ACM HotNets, 2020. [29]
- Simon Kassing and Ankit Singla. *CodeBind: tying networking papers to their experiment code.* 2021. [109]

* Shared first author.

2 NEW PRIMITIVES FOR BOUNDED DEGRADATION IN NETWORK SERVICE

We pose that there is value in further exploring network primitives which are flexible rather than strict. The principle idea is that applications which have less strict requirements can cede their quality of service to improve or accelerate those which are less flexible. We consider in particular two dimensions of interest: partial delivery and bounded deprioritization. First, in terms of partial delivery, current common network primitives are insufficient: TCP offers only fully reliable transmission whereas UDP provides no guarantees. Recent work in the ML research community [210] has shown for instance that network loss to a certain extent has minimal effect for many types of ML training. Another work, ATP [129], has explored partial delivery, but not the co-design of both loss and deprioritization guarantees, and rigorously defining when data should be considered lost. Second, although network level mechanisms such as prioritization and scheduling provide a way to assign quality of service, they do not offer bounded deterioration. There do already exist approaches to enable non-competing background traffic such as LEDBAT [117], which however similarly do not provide any guarantees to its deprioritized traffic.

We set out to design network primitives capable of both (a) ensuring that regular traffic is serviced preferentially, and (b) the extent of this preferential treatment is bounded such that the corresponding deterioration in network service for flexible traffic is controlled, thus only causing only negligible or acceptable application-level performance impact. In particular, we explore the co-design of both loss and deprioritization guarantees, and

when data should be considered lost.

We take the first steps in this direction by: (a) introducing primitives describing partial delivery and bounded degradation goals; and (b) designing mechanisms for the network that implement these primitives. We propose a simple extension to the application-network interface: an application sending data can specify what fraction of it must be reliably delivered, and/or what degradation in performance is acceptable compared to the default fair-sharing of the network. Using this interface, different applications (*e.g.,* different ML training approaches and workloads) can specify arbitrarily different degrees of acceptable degradation in the same network. We also design and implement algorithms for the network to exploit the headroom these relaxations of the network service objectives provide, by dropping or deprioritizing flexible traffic in favor of regular traffic, while ensuring that the specified (relaxed) guarantees for flexible traffic are indeed met.

We have evaluated our approach, **ReFlex**, using packet-level simulation. We show that by allowing (guaranteed) partial delivery and bounded deprioritization for flexible traffic, the performance of regular traffic can be improved. We compare our approach with standard prioritization methods, and investigate the extent to which both are able to bound degradation as well as speed-up workloads.

Chapter outline. In this chapter, we first motivate the need for new network primitives and formalize the reliability and aggressiveness factor in §2.1. Second, we introduce the abstract design of our algorithm in §2.2. Third, we describe the practical probing algorithm design in §2.3. Fourth, we describe the implementation in §2.4. Fifth, we evaluate our approach in a diverse set of setting using packet-level simulation in §2.5. Finally, we discuss future work in §2.6, related work in §2.7, and provide an overall summary in §2.8.

Contributions. Text and figures from the following publication were included in this chapter and its corresponding introduction and conclusion:

- Simon Kassing, Vojislav Dukic, Ce Zhang, and Ankit Singla. *New primitives for bounded degradation in network service.* arXiv:2208.08429, 2022. [107]

In the collaboration, both I and Vojislav Dukic worked on the conceptual idea and theory, with me having an additional focus on the experimental evaluation.

2.1 Why new networking primitives?

Network traffic for a growing class of applications offers two degrees of flexibility that cannot be exploited by today's networks: (a) such traffic does not need the predominant reliable delivery of TCP; and (b) such traffic can be deprioritized in favor of more other time-critical traffic. However, providing *no* network service assurance is also unacceptable, as that can lead to substantial application-level performance degradation. We next discuss each of these potential relaxations to the network service model in greater detail.

2.1.1 Guaranteed partial delivery

Traditionally, networking has provided only two extremes: (1) a fully reliable in-order data delivery primitive, using TCP or its derivatives; and (2) entirely best-effort data delivery using UDP, with no reliability guarantees. While many existing applications depend on one of these service models, new emerging workloads can benefit from guaranteed partial reliability. For instance, in certain types of distributed machine learning training, losing some fraction, *e.g.*, 20% of each model update transferred between workers, or between workers and a parameter server, has no or negligible impact on the ML training task [210]. Note that this is different from adaptive applications like video streaming, where there is first a compromise in application performance by picking lower video quality, and then data for the lower quality still needs to be delivered reliably using TCP, or is entirely best-effort, again with additional observable degradation.

To characterize flexible traffic that only requires guaranteed partial delivery, we define a reliability factor as follows:

Definition 1. *Each flow f has a **reliability factor** $r \in [0, 1]$, which is a lower bound on the fraction of its data that must be delivered reliably by the transport protocol. For regular, inflexible traffic, and more broadly, for any traffic for which r is not explicitly specified, $r = 1$ by default.*

A naive way of exploiting the flexibility specified using $r < 1$ would be to simply trim the payload before each transfer, for instance, transfer the first 80% bytes for each model update reliably using TCP, and then drop the last 20% bytes. While this is indeed "partially reliable" delivery, this is a poor strategy: what if the network was congested earlier in the transfer, but is under-utilized later? In this case, sending less data does not

have much upside in terms of improving network multiplexing. Thus, we would like to use this flexibility adaptively, dropping data when the network is busy with regular, inflexible traffic, while delivering it when the network is less congested. The knowledge of payload size in advance is fundamental to partial delivery: if the control mechanism only knows the final payload size once the transfer is complete, it is unable to finish early before delivering the entire payload.

Unfortunately, today's binaries of reliable and unreliable transfer do not accommodate such sophisticated, adaptive switching between reliable and unreliable transmission. Interpreted in light of the above definition, using UDP effectively implies $r = 0$, while using TCP enforces $r = 1$. Note that our definition does not enforce that only r fraction of traffic is delivered, rather r is a lower bound, with at least r fraction of traffic sought to be reliably delivered. The need for a new partially-reliable transport protocol has been recognized by a parallel work available as an online manuscript in which the Approximate Transmission Protocol (ATP) has been proposed [129]. ATP defines a similar reliability factor and provides guarantees for the data delivered. The key distinction is that ATP decides partial delivery based on actual packets being lost, which differs from our approach which defines partial delivery based on bandwidth loss relative to fair share (§2.2). More distinctions to ATP are further explained in §2.7.

2.1.2 Bounded deprioritization

Many applications, such as backup or administrative tasks, non-interactive analytics, and ML training workloads where real-time model freshness is not critical, can be bandwidth-hungry, but do not necessarily need to compete fairly with more time-sensitive traffic such as user-facing Web search or ML inference queries. Networking does offer three broad types of primitives to benefit regular traffic by exploiting such flexibility, but as we discuss in the following, none of these provides any bounds on how much worse the performance of the flexible traffic can be.

Background transport: LEDBAT [117] attempts to capture unused network bandwidth, and is commonly used over the Internet today for transfers like software updates. However, LEDBAT does not compete fairly with TCP traffic at all, and thus, if used for flexible traffic, cannot bound the degree to which flexible traffic is deprioritized: if the network is seeing a large volume of regular traffic, flexible traffic using LEDBAT would be completely starved.

Prioritization: Superficially, running flexible traffic at low priority using multiple queues available in most commodity switches today is a natural way of favoring regular traffic over flexible traffic. A variety of more sophisticated scheduling schemes are also known that allow multiple degrees of low priority [8] for different degrees of flexibility in traffic, or weighted prioritization (instead of absolute) with configurable weights [48]. However, regardless of the details of how prioritization is implemented, it offers no guarantees: the performance of deprioritized traffic can be arbitrarily worse than its fair-share of network bandwidth. If there is a surge in high priority traffic, low priority traffic suffers starvation. While starvation can be somewhat reduced using "aging", whereby a flow's priority increases over time [41], this is a heuristic with no guarantees. A broader issue is that priorities are meaningful only relative to other network-wide traffic's priorities and volume, which an individual sender does not know. Thus, even weighted prioritization results in no guarantees: if traffic with many different degrees of flexibility co-exists in a network, each class of flexible traffic has no knowledge of what share of bandwidth its weight will translate to, given that other weights are unknown to it.

Deadline-aware networking: In a similar vein to priorities, one can also specify explicit deadlines for different traffic flows to finish [195]. Deadlines exhibit some of the same problems as noted for priorities above. In addition, setting deadlines is especially difficult for flows that have soft deadlines, where the performance degrades gradually with time, and it is unclear what to do when a deadline is not met. The flexible workloads we describe are of this type, *e.g.,* even if there is no explicit deadline of when an ML model must finish an update, delaying updates gradually increases staleness, and degrades performance. While one can potentially set deadlines for other traffic and use leftover bandwidth for flexible traffic [41], this provides no guarantees to flexible traffic. As work on deadline-aware scheduling notes, such workloads must be managed using heuristics [203], with no guarantees.

Thus, none of the existing networking primitives allow the specification of a bounded degree of deprioritization, where a guarantee can be specified such that it holds independent of other traffic. Further, primitives to bound starvation can actually cause flexible traffic to not back off appropriately (*e.g.,* with weighted prioritization) or worse, increase in aggressiveness (*e.g.,* with aging) at unfortunate times when the network is most congested.

Given the above discussion on the limitations of existing primitives, one may even

ponder whether any guarantee on bounded deprioritization is possible to frame and achieve at all without explicit reservations, which are unsuitable for our purposes of degraded service. We find that while absolute guarantees are unachievable simply because other traffic is variable and can congest the network, a guarantee can be framed relative a flow's fair-share bandwidth. To do so, we define how aggressive a flexible flow is as follows.

Definition 2. *Each flow f has an* ***aggressiveness factor*** $\alpha \in [0, \infty)$*, which specifies the fraction of f's max-min fair-share bandwidth that will be guaranteed for f over its transmission. For regular, inflexible traffic, and more broadly, for any traffic for which α is not explicitly specified, $\alpha = \infty$ by default.*

Like our discussion of the reliability factor above, α is only a lower bound, with a flow seeking at least that fraction of its fair-share bandwidth. TCP, by targeting (but not always achieving) max-min fair-share bandwidth, effectively enforces $\alpha = 1$. Flexible flows with $\alpha < 1$ thus cede capacity to regular flows, as well as flexible flows with higher α values. In certain cases with other flexible flows present, a flow might actually achieve a rate higher than their max-min fair share. An α value greater than 1 specifies how much rate speed-up a flexible flow permits. Regular flows want the maximum amount of rate speed-up fairly available, as such for them by default α is set to ∞. In practice an $\alpha > 1$ is generally unable to be guaranteed as it depends on other flows ceding their fair share (*i.e.*, having their α less than 1).

Note that our above framing allows flexible flows to specify a bounded degree of deprioritization compared to their default TCP-driven fair-share state. This is particularly attractive for applications with soft deadlines — there is no explicit notion of a hard deadline, but performance degrades with time, so specifying an $\alpha < 1$ allows specifying how much degradation compared to the default case is acceptable. Indeed, the fair-share depends on other network traffic and varies over time, but it can be independently estimated by regularly probing the network (such a mechanism is described §2.3.1).

2.2 Abstract algorithm design

2.2.1 Budgeting algorithm for bounded degradation

The key challenges in enforcing bounded degradation stem from the lack of information:

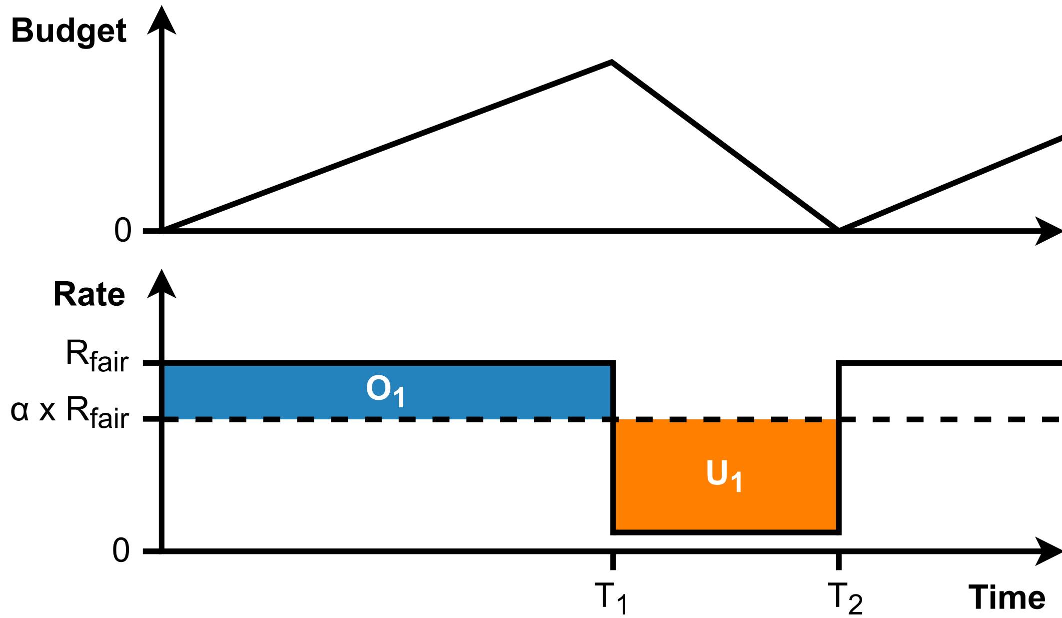


Figure 2.1: The algorithm is based upon collecting budget and once it has been built up (O_1), having the budget be spent to prioritize regular (more time-critical) flows (U_1). Budget can be built up through α by having a rate $> \alpha \times R_{\text{fair}}$ or through r in a similar way.

1. A flow's size may not always be known at its start, and as it may finish anytime, we must ensure that it has delivered at least r fraction of its data at *every instant*.

2. As flows arrive and finish, a flexible flow's fair share bandwidth continually evolves, and it is unclear when to incur deprioritization such that the α-bound holds.

REFLEX addresses these challenge by building up a budget for degradation of flexible flows and then exploiting it. Flexible flows receive normal service for a period such that they run ahead of their performance goals (growing the budget), and then are degraded opportunistically when needed to speed up regular flows (depleting the budget). For now, we assume that the fair share for each flow is known to it at all times. We shall later incorporate probing to estimate the fair share.

In the following, we shall illustrate this budget buildup and exploitation in the context of flexible flows that seek fully reliable delivery ($r = 1$), but can tolerate bounded deprioritization ($\alpha < 1$). The logic for flows that allow partial delivery but not deprioritization ($r < 1$, $\alpha = 1$) is similar. For flows that allow both, deprioritization is used first because

in any case, low-priority flows cede capacity to regular flows; loss is invoked after the deprioritization budget is fully depleted.

Regardless of the number of different flows with different degrees, α_i, of deprioritization permissible, REFLEX uses two priority queues at each switch. Commodity switches used today typically provide at least 8 priority queues [45].

We explain the algorithm for switching between high (default, used for regular flows) and low priorities visually using Fig. 2.1. A flexible flow starts by fairly competing with other flows at high priority, and receives its fair-share rate R_{fair}. Until time T_1, it sends S_1 bytes at this rate. Since we need only guarantee $\alpha \cdot R_{fair}$ bandwidth, this implies that $(1-\alpha) \cdot S_1$ more bytes than needed to meet the guarantee are delivered by T_1. This is the area marked O_1 in Fig. 2.1. These "overdelivered" bytes represent the accumulated budget.

Once sufficient budget is accumulated, the flow switches to low priority, ceding capacity to other flows, until time T_2. At low priority, the flow may potentially receive less than its fair share of bandwidth, thus sending fewer bytes than it could have at the fair share. These "underdelivered" bytes, shown in area U_1 in Fig. 2.1, are a depletion of the budget. Given our assumption of knowing the fair share at any time, we can always calculate the depletion in the budget and ensure that the budget is always non-negative. When the budget is completely drained, the flow is switched back to high priority for rebuilding its budget.

Note that in the above, the time when the flow switches back to high priority, T_2, is obvious: whenever the budget reaches zero. However, T_1 is a design parameter: how much budget to accumulate before switching to low priority? If T_1 is small, a flow will switch more frequently, as it builds up and expends buffer. In our abstract description thus far, this is immaterial, but in a practical system, there is a convergence period in which a flow arrives at its new bandwidth allocation each time such a switch takes place, and T_1 must be set such that it is several times the convergence time. We provide a detailed evaluation of this parameter in §2.5.4.

We make several remarks about this simple algorithm:

1. Budget building must only use past history, as any assumptions on future bandwidth can be mistaken, and violate the guarantee of α.
2. For partial delivery, a flow can finish anytime, and must have sent r fraction of its

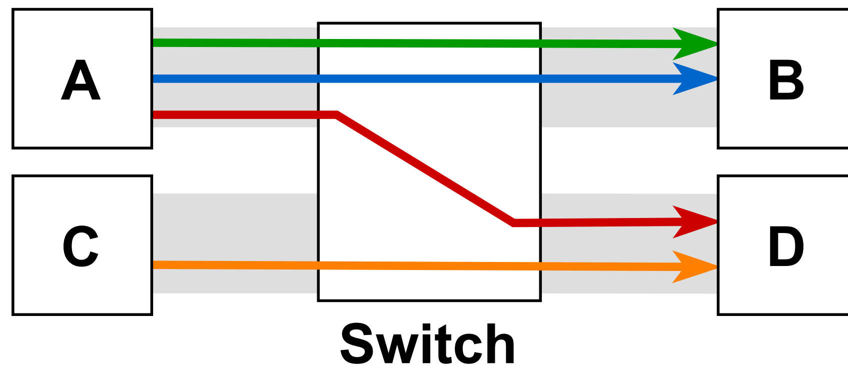


Figure 2.2: A max-min fair allocation system in which removing one flow between A and B will degrade the performance of the orange flow between C and D.

data *at every instant*. As flow size being known in advance is fundamental to partial delivery, the loss budget is known too, and can be used to improve performance (§2.5.3).

3. Low priority can, but does not always drain the budget; the budget is drained only while a flow receives bandwidth smaller than $\alpha \cdot R_{fair}$.

2.2.2 Max-min guarantee in multi-hop

Up until now, we have primarily considered the scenario of flows sharing a single link, in which max-min fairness is straight-forward to calculate and reason about. However, in a max-min fair allocation system, downstream changes have upstream effects. More specifically, removing some traffic (or moving it to low priority) on one path can cause performance *degradation* on another.

Consider an example with 4 hosts attached to a switch, as shown in Fig. 2.2. Two flows are going from A to B, one from A to D and one from C to D. If all the flows are at the same priority (default), flows that originate from A have 1/3 units of bandwidth, while the orange C-D flow utilizes 2/3 units of bandwidth. If one flow from A to B switches to low priority, e.g. the top (green) flow, it will receive 0 units, while all other flows would get 1/2, including the (orange) one between C and D. This further implies that, the regular C-D flow experiences a slowdown of 0.25 compared to the default, just because a flexible flow on another path switches to low priority.

Note however, that the above is not a violation of our bounded deprioritization guar-

antee: for any flow, it gets α fraction of its fair-share bandwidth, but due to the nature of max-min fair allocation, the fair share can itself change as other flows shift their priorities. This is precisely why we do not state (and cannot fulfill) guarantees framed in terms of flow completion time. The above discussion is meant to highlight that the fair-share-degradation guarantee is weaker than an FCT-degradation guarantee, and REFLEX only provides the former. The fair-share-degradation guarantee of a flow is only as strong as the fair share it can achieve given the priorities of other flows, which thus can be less than its fair share when all flows are of equal priority.

2.3 Practical algorithm design

The abstract budgeting algorithm as described in §2.2 suffices to achieve guaranteed partial delivery regardless of a real transport implementation's quirks, as long as fully reliable delivery is correctly implemented using acknowledgments and retries. However, the bounded deprioritization guarantee relies on two assumptions: (a) each flow knowing its fair share at all times; and (b) the transport protocol converging to fair share instantly after flow arrivals, departures, and priority switches. In practice, these assumptions do not hold. For (b), the transport protocol's degree of divergence from fair share is, of course, inherited by REFLEX. For (a), in our approach we choose to have each flow probe the network periodically to approximately estimate its fair share, as we describe next. Having only approximate estimates for fair-share bandwidth, and incurring some time for convergence to the fair share, both lead to imprecision in the α-bound, which we later explore experimentally.

2.3.1 Probing mechanism

We propose the usage of regular probing for each flexible flow to determine its fair share. The key challenge is estimating a flow's fair share through probing is that all flows must operate approximately in synchronous fashion at high priority in the probing period, such that each flow can independently arrive at its fair share estimate. We rely on a data center wide synchronization primitive for this purpose. While recent work has claimed data center wide synchronization within tens of nanoseconds [63, 123], for our purposes, microsecond-scale synchronization using PTP [128] is sufficient. Note that

while such synchronized probing may sound aggressive, it is merely a brief return to today's default behavior, with more operational time spent using the results of probing.

Each flexible flow operates in a (synchronized) cyclic fashion with three phases: (a) *warmup*, with all flexible flows at high priority; (b) *measure*, with all flexible flows at high priority; and (c) *exploit*, where each flow individually decides whether to switch to low priority. The warmup phase allows time for convergence to fair share, before a fair-share estimate is made from the measure phase. We define the following parameters:

- T_{int}: interval duration (ms)
- D_{warmup}: warmup phase duration (no. intervals)
- $D_{measure}$: measure phase duration (no. intervals)
- $D_{exploit}$: exploit phase duration (no. intervals)

Once the fair share estimate is available after the measure phase, each flow calculates the amount of potential expense (*i.e.,* budget loss) of going low priority in the upcoming phase. This expense amounts to $\alpha \times R_{fair} \times D_{exploit} \times T_{int}$. Only if the current budget is larger than the expense, does the flexible switch to low priority for the exploit phase.

The fair share estimate which is established in the measure phase is used throughout all subsequent intervals to accumulate or drain budget. As such, although a flexible flow can only switch to low priority during the exploit phase, it collects budget in all phases (with the exclusion of the first partial cycle it needs to synchronize up).

Note that explicit coordination is not required across flows: the lengths of the warmup, measure, and exploit phases are set network-wide. A newly started flexible flow starts at high priority until the next synchronized cycle starts, for which its uses its own (network-wide synchronized) clock. It can calculate at any time which part of the cycle is currently active using one simple modulo operation. If the timesteps are on the order of hundreds of microseconds or more, microsecond-scale synchronization is clearly sufficient. Additionally note that if transport protocol convergence were faster, *e.g.,* by approximating RCP in modern programmable switches [180] or implementing PERC [101], we could eschew the warmup phase entirely, thus increasing the fraction of time flows can operate in the exploit phase.

2.3.2 Algorithm

Budget adjustment algorithm. The core of the algorithm is the manner in which it accounts budget, which is abstractly described in §2.2. The algorithm requires a fair share estimate R_{fair} to account for the budget accumulation or drain. The budget must be updated at a time granularity fine enough to (a) incorporate the frequency at which the fair share estimate is renewed, and (b) before a priority decision is made. In the probing mechanism, this is performed at T_{int} interval granularity, which meets both criteria. In the adjustment algorithm, two budgets are maintained:

- Rate degradation budget B_α. It increases each update when the actual rate R_{actual} is higher than $\alpha \times R_{fair}$.
- Reliable delivery degradation budget B_r. With the assumption that the flow size F is known in advance, B_r starts at the $(1 - r) \times F$. Only if $r < 1$ is the flow size required in the algorithm. We leave exploration of other (later) signaling of flow size to future work.

The budget is decreased by first draining the rate degradation budget B_α, and only if that is empty, draining B_r. The reasoning behind this is that draining B_r results in less flow data being sent out, which is undesirable if the targeted fair share portion is still being achieved. The B_α can become negative if consistently we achieve lower rate than intended – which in turn will trigger going at high priority to remedy this (see next section). The budget algorithm pseudo-code is depicted in Alg. 1.

Control algorithm with probing. The control algorithm has four tasks: (1) it determines whether the flow is finished earlier due to reduced reliability, (2) it awaits to be synchronized with the other flexible flows, (3) after a complete warmup and measure phase, it calculates the fair share which is used in the next cycle, and (4) it decides whether to go at low or high priority during the exploit phase. In each update, the current interval is identified with ID_{int}, which is determined using a modulo operation of the clock and the interval duration T_{int}. The control algorithm pseudo-code is depicted in Alg. 2.

Algorithm 1 Budget adjustment logic for each elapsed time period of the flow in which the fair share rate R_{fair} is known.

Initial state:
F: flow size (known if $r < 1$, else ∞), $S_{sent} = 0$, $B_\alpha = 0$, $B_r = (1 - r) \times F$

```
1:  function ADJUST_BUDGET(ACK_byte, T_elapsed)
2:      R_actual = ACK_byte / T_elapsed
3:      S_sent = S_sent + ACK_byte
4:      B_α = B_α − T_elapsed ∗ (α × R_fair − R_actual)
5:      if B_α < 0 then
6:          B_r,before = B_r
7:          B_r = B_r + B_α
8:          if B_r >= 0 then
9:              B_α = 0
10:         else
11:             B_α = B_r
12:             B_r = 0
13:         end if
14:         S_sent = S_sent + (B_r,before − B_r)      ▷ B_r drainage reduces how much to send
15:     end if
16: end function
```

2.3.3 Probing consequences and limitations

The introduction of probing to periodically determine the fair share has certain consequences and limitations.

What is the effect of an inaccurate fair share estimate? An inaccurate fair share estimate can either be too low or too high. In the former (too low estimate), the budget will increase too much as the gap $R_{actual} - \alpha \times R_{fair}$ is larger than in actuality, leading potentially to an inflated budget and thus a rate which can deteriorate beyond the theoretical guarantee. In the latter (too high estimate), the budget will not build up as much, thus resulting in less flexibility. An inaccurate estimate can be due to several factors. Firstly, it takes time for a congestion control protocol to converge, which can lead to inaccuracies. This can be ameliorated by having large enough phases, as well as the choice of a quickly converging congestion control protocol. Secondly, the measured fair share can be out of date because new flows can arrive or existing ones depart. This is especially the case when there are flows present whose completion time last shorter than a full cycle, which would lead to a reduced rate during whichever phase it is active.

Algorithm 2 Control algorithm with probing. The update is called at the T_{int} interval with an incremented ID_{int} each time. The ID_{int} is determined using the global synchronized clock each flexible-capable host maintains.

Initial state:
R_{fair} = **not set**, P_{prev} = NONE, SYNC = **false**, $D_{probe} = 0$, $T_{probe} = 0$, $I_{exploit\ low}$ = **false**

```
1:  function UPDATE(ID_int, ACK_byte, T_elapsed)
2:      if R_fair is set then
3:          ADJUST_BUDGET(ACK_byte, T_elapsed)
4:      end if
5:      if S_sent ≥ F then
6:          return FINISHED
7:      end if
8:      if P_prev == MEASURE and SYNC then
9:          D_probe = D_probe + ACK_byte
10:         T_probe = T_probe + T_elapsed
11:     end if
12:     P_upcoming = DETERMINE_PHASE(ID_int, D_warmup, D_measure, D_exploit)
13:     if P_prev == EXPLOIT and P_upcoming == WARMUP then
14:         SYNC = true
15:         I_exploit low = false
16:     end if
17:     if P_prev == MEASURE and P_upcoming == EXPLOIT and SYNC then
18:         R_fair = D_probe / T_probe
19:         D_probe = 0,  T_probe = 0
20:     end if
21:     L_potential = α × R_fair × D_exploit × T_int
22:     if P_prev == MEASURE and P_upcoming == EXPLOIT and R_fair is set then
23:         I_exploit low = true if B_α + B_r > L_potential else false
24:     end if
25:     P_prev = P_upcoming
26:     return LOW if I_exploit low else HIGH
27: end function
```

Flexibility limited to relatively large flows. We acknowledge that our service degradation primitives are only meaningful for relatively large flows, which run for at least tens of milliseconds. We believe this to be reasonable: there is, after all, little advantage from degrading service for short flows that don't consume most of the network's capacity. However, by degrading service for a few such large flows, we can benefit a large number of short flows, which typically have more stringent service requirements.

Speed-up limited to exploit phase. In the warmup and measure phases the flexible flow is unable to be at low priority. As a consequence, the maximum fraction of time a flexible flow can be at low priority is $D_{exploit}/(D_{warmup} + D_{measure} + D_{exploit})$. This is especially impactful if there are short regular flows, as if they occur in the first two phases they will receive only fair share rather than be sped up. This can be ameliorated with a relatively longer exploit phase, which in turn however would reduce the fair share estimate accuracy.

Time synchronization required among flexible hosts. Participating hosts which wish to participate in the starting of flexible flows must be time synchronized. As such, the deployability of REFLEX is limited to those settings.

2.4 Implementation

For ease of understanding, we have thus far described REFLEX at an abstract, algorithmic level. Implementing it requires small changes at multiple levels of the networking stack, which we discuss in a top-down manner.

Application changes: Applications that wish to send flexible traffic must correctly use an extended sockets API to specify the degradation parameters for their traffic. An application using only deprioritization does not see any change in terms of transport behavior, except performance. However, applications allowing partial delivery might see data loss, and thus must prepare data correctly such that partial deliveries are usable.

REFLEX must be able to discard data if reduced reliability is permitted. When REFLEX decides to discard some data segments, it will remove these data segments from the send buffer. As such, application have to ensure that *any* data segment transmitted over a flexible socket can be removed from the flow without hindering the receiver from being able to make use of the rest of the data segments. REFLEX must similarly be informed of the application data segmentation strategy, which could range from a static constant (*e.g.*, every B bytes is an individual data segment) to dynamically informed strategies. This would incur a header overhead for each data segment. In this work, we assume that data is segmentable at the byte level. We leave investigation of data segmentation and discard decision strategies to future work.

Application-transport interface changes: The sockets API has to be extended to support

the creation of flexible flows. For every flexible flow, the application must be able to specify the parameters α and r through the socket API. By default, α is set to ∞ and r is set to 1, such that legacy applications do not need to change, and can continue operating as regular traffic, while benefiting from flexible traffic's behavior.

Transport level changes at hosts: Our design requires that the transport implementation be extended with the REFLEX state machine. For each flexible flow, it must track the budget and the time slots (warmup, probe, exploit) using a system time synchronization protocol like PTP, perform fair-share measurements, decide on whether to operate in low or high priority or / and discard or retransmit data, and tag packets with the appropriate priority levels.

REFLEX benefits if the transport protocol converges fast to fair-share bandwidth. Thus, in our implementation, we use DCTCP instead of TCP. DCTCP, by making use of congestion markings (ECN), can make more accurate rate adjustment decisions than TCP. We use DCTCP because it is already commonly used in data centers, but if RCP or PERC were implemented to achieve even faster convergence, REFLEX's performance would improve further.

Network-level changes at switches: REFLEX's bounded deprioritization uses 2 priority queues, while commodity data center switches commonly support 8 queues. The remaining available priority levels can still be utilized for explicit prioritization outside the domain of REFLEX, *e.g.,* to prioritize network control messages.

An alternative to such in-network prioritization is to simply have "wimpier" congestion control logic kick in when flows are operating in degraded mode, *e.g.,* back off by a larger multiplicative decrease factor on packet loss than regular flows. We have not explored such a strategy yet.

2.5 Evaluation

We evaluate REFLEX using packet-level simulations. We first explain the experimental setup including network devices, congestion control, and default parameterization in §2.5.1. Next, we show the motivation of REFLEX by comparing it in a simple scenario against other prioritization schemes in §2.5.2, and showcase how partial delivery works in §2.5.3. We explore the parameterization of the probing mechanism in §2.5.4. Finally,

we run REFLEX for larger workloads to examine the utility that flexible flows offer and compare it to the performance of fixed weighted prioritization in §2.5.5.

2.5.1 Experimental setup

For the packet-level simulations we make use of ns-3 [152] with the basic-sim module [1, 60, 104, 106]. The default setup is as follows (unless otherwise specified in the upcoming sections). The network consists of 10 Gbit/s links with a delay of 1 μs. Each network device has two traffic control queue discipline queues, with a weighted scheduling mechanism which it draws from using deficit round robin. The high priority queue has 90% of bandwidth, and the low priority 10% of bandwidth. Both traffic control queues are configured with Random Early Detection (RED) as its queue discipline, with a hard marking threshold of 65 packets and a maximum size of 466 packets (both following recommendation from [6]). Each network device finally has a simple tail drop queue of 20 packets. Network devices use an MTU of 1500 byte.

We use DCTCP as the congestion control protocol. The various timeout values have been adjusted to be effective in a low latency (2-4 μs base RTT), high throughput (10 Gbit/s) environment, in particular the minimum RTO is set to 1 ms, the initial RTT measurement to 2 ms, the connection timeout to 2 ms, the delayed ACK timeout to 1 ms, the persist timeout to 8 ms and the maximum segment lifetime to 8 ms. The clock granularity is set to 1 ms. The segment size is set to 1380 byte. It is not possible to have the throughput match the theoretical line rate due to the overhead of the point-to-point, IP and TCP (with TS option) headers. As a result, approximately 96% of line rate is the maximum throughput. The send and receive buffers are set to 1 MiB.

REFLEX is by default configured with $T_{int} = 5$ ms, and with $D_{warmup} : D_{measure} : D_{exploit}$ set to 1 : 1 : 3. This matches the low latency and relatively high throughput setting. Parameterization is further explored in §2.5.4.

2.5.2 Prioritization scheme comparison

Use cases. The difference of REFLEX to the prioritization alternatives described in §2.1.2 is best illustrated by an example. Consider a single link with a bandwidth of 10 Gbit/s over which two flows are started. We bring forward two scenarios, the first to showcase speed-up and the second to showcase starvation. In both cases, the flexible flow wants

its performance to be degraded by at most 10% of its fair share ($\alpha = 0.9$). The regular flow expects usual performance and thus does not want its performance degraded.

- Case 1: 40 Gbit flexible flow starts at T=0 s, and a 2 Gbit regular flow starts at T=2 s
- Case 2: 40 Gbit regular flow starts at T=0 s, and a 2 Gbit flexible flow starts at T=2 s

Results. In the baseline setting (*i.e.,* without any prioritization scheme), the flows compete fairly and thus in both cases the flexible and regular flow achieve 5 Gbit/s when both are active. When using absolute prioritization, the flexible flow achieves 0 Gbit/s when both are active, and the regular flow 10 Gbit/s. Thus, in Case 2 this result in the flexible flow being finished only after the regular flow is done. For weighted prioritization, we consider three variants for comparison: regular flows having 9x more weight, 2x more weight, and $1\frac{1}{9}$x more weight. Among these three variants, either there is little speed-up of the regular flow in Case 1, or starvation of the flexible flow in Case 2. In Case 1, REFLEX builds up budget in the first 20 s which means that it can be on low priority when the regular flow arrives, giving it all the high priority bandwidth. However, because of the presence of probing with 1:1:3 phases, the high priority can only be consumed 60% of the time. This is shown in Fig. 2.5 in the changing between the fair share at high priority and the reduced rate at low priority. Thus, the regular flow rate calculated rate is 7.4 Gbit/s. In Case 2, REFLEX make sure the flexible flow to maintain the $\alpha = 90\%$ guarantee, therefore not permitting the regular flow to degrade

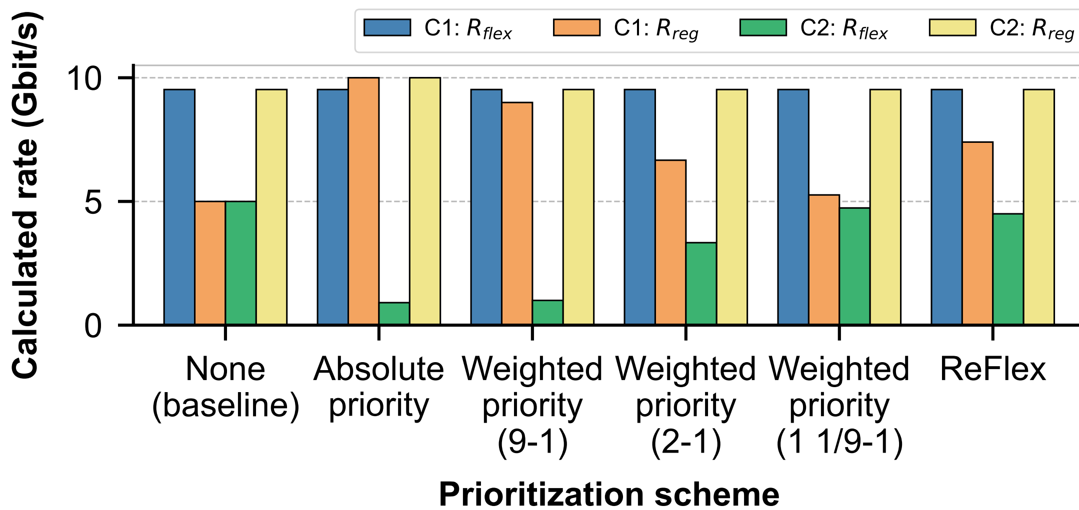


Figure 2.3: Calculation of different prioritization schemes for the two cases.

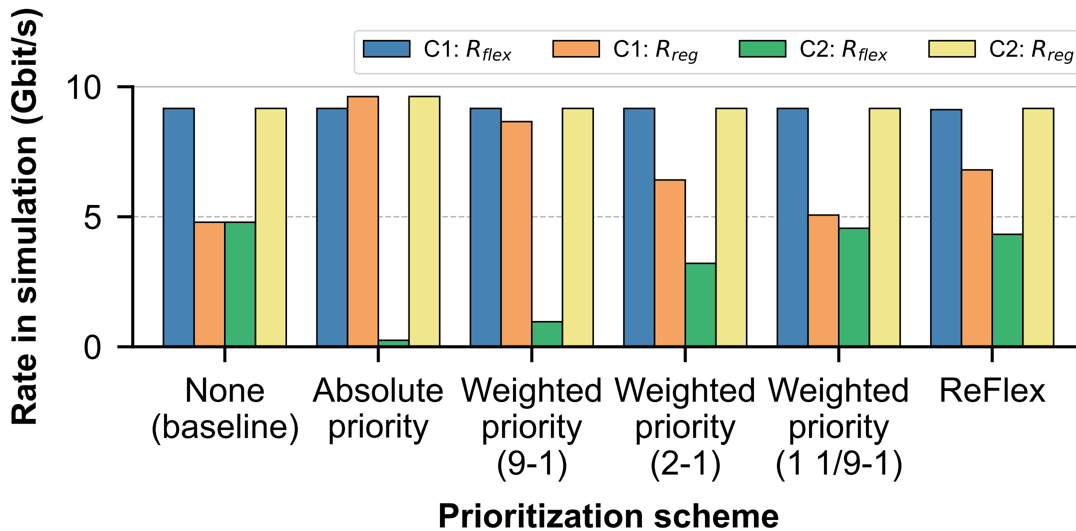
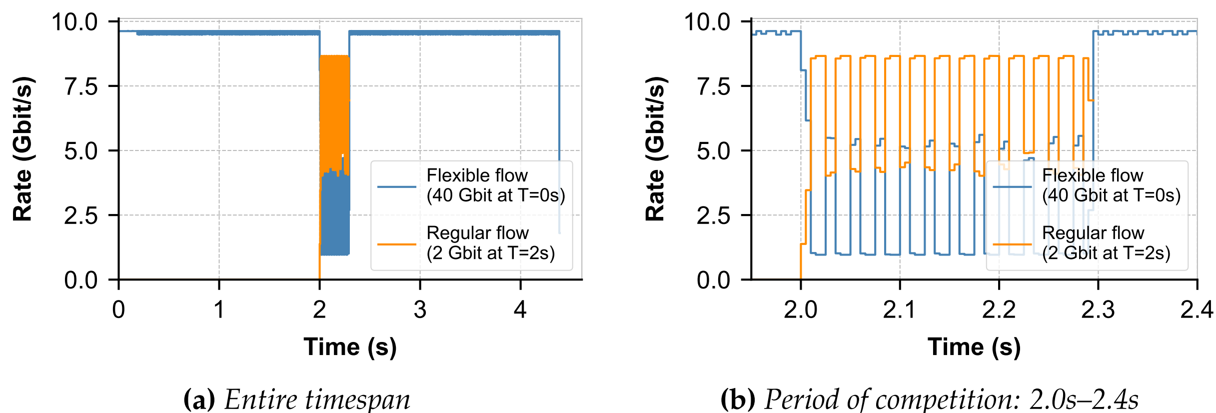


Figure 2.4: Packet-level simulation results of different prioritization schemes for the two cases.

the flexible flow's rate too much. This yields a rate of 4.5 Gbit/s for the flexible flow. An overview of the average rates achieved is shown in Fig. 2.3 for calculation. Barring the header overhead, the calculation results match those of the simulation in Fig. 2.4. Note that in Fig. 2.5 there is a small rate fluctuation for the flexible flow, which is caused by retransmission due to reordering. Among the compared prioritization schemes, REFLEX is able to (a) achieve a significant speed-up of the regular flow in Case 1, and (b) prevent starvation of the flexible flow in Case 2 comparable to the baseline. This is evidently workload dependent: if flexible flows are not starved, other fixed (weighted) prioritization schemes perform better.

It is possible to achieve better speed-up in Case 1 through either increasing weight of the high priority queue, or, more effectively, by changing the ratio of the probing phases to have a relatively longer exploit phases. The former has as a side effect that congestion control protocols do not respond well to complete starvation, as they will be deprived of any network signals. The effects of the latter are further explored in §2.5.4.

Different from deadlines. The most distinguishing characteristic of REFLEX is that it defines its guarantees in relation to the network state, in particular the fair share each flow should achieve. This is distinctly different from explicit deadlines, which are set agnostic to network state. Indirectly, an absolute time deadline corresponds to a certain desired rate provided the data transfer size. A simple approach would be to set these



(a) *Entire timespan* **(b)** *Period of competition: 2.0s–2.4s*

Figure 2.5: For Case 1, REFLEX yields considerable speed-up for the regular flow, although it its effect is reduced due to the warmup and measure phases.

absolute time deadlines based on the line rate. With this approach, for Case 1 one would set the deadline of the flexible to T=4.4 s (4 s + 10% for flexibility) and of the regular to T=2.2 s, and in Case 2 of the flexible to T=2.22 s (0.2 s + 10% for flexibility) and of the regular to T=4 s. Case 1 would be possible for the deadlines to be met, however in Case 2 it is not feasible. Of course, one can also have or set more relaxed deadlines, which still nevertheless is based on an assumption of the number of present flows and their target rate. The deadline-aware mechanism, both centralized scheduler such as FastPass [159] and decentralized such as D^{2}TCP [195], must decide how to handle infeasible deadlines as well as how to accommodate the non-deadline-aware traffic.

Priority scheme comparison takeaways: REFLEX makes use of probing to prevent a flexible flow from being starved. It is in low priority as long as it cumulatively receives the portion of fair share it requires. The potential benefit of REFLEX is (a) *workload dependent* because it requires flexible flows to be starved to outperform other prioritization schemes, and (b) *objective dependent* as it is useful only if it is desirable in moment of high utilization to not fully prioritize regular flows.

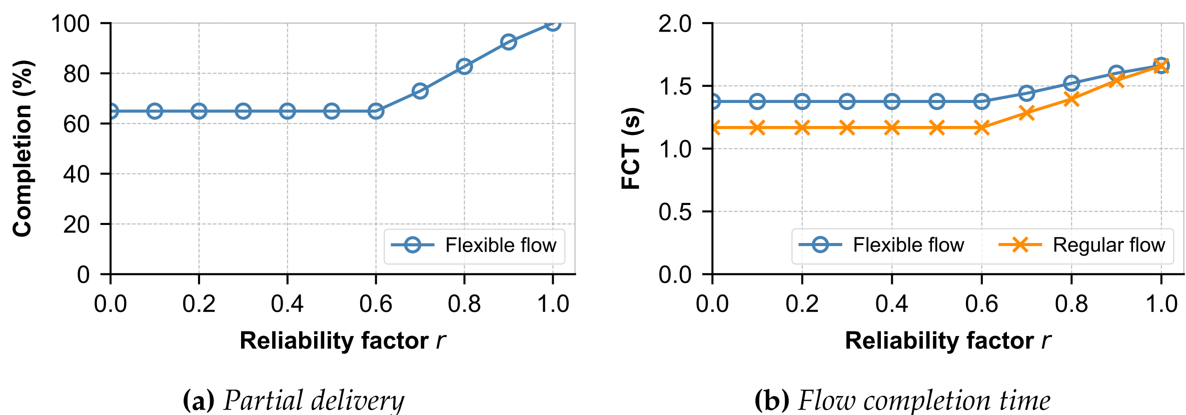
2.5.3 Decision of partial delivery

In this section, we turn our focus to partial delivery. For flexible flows which permit partial delivery, REFLEX makes the decision whether to not deliver part of data in the same manner it drains the budget: through the difference between the actual rate and

the fair share rate multiplied with aggressiveness factor α. Through going at low priority, the algorithm permits other (regular) flows if present to reduce the budget and in turn the portion of data delivered by the flexible flow. Unlike α, a flow actually immediately starts with a budget with the flow size known in advance.

We showcase with a single flexible flow of 8 Gbit competing against a regular flow of the same size. We vary the reliability factor r from 0 (no reliability guarantee) to 1 (complete reliability guarantee). The results are shown in Fig. 2.6. The competitive period lasts until the regular flow finishes. Up until around $r = 0.6$, the lower the reliability factor r the less data of it is delivered (Fig. 2.6a), and thus simultaneously the faster the flexible completes itself (blue line in Fig. 2.6b). Similarly, the regular flow is significantly sped up as the flexible yields the bandwidth to it (orange line in Fig. 2.6b). The ability of the flexible flow to degrade is limited by the bandwidth it achieves at low priority, which is at most 10% of bandwidth in the current setting if there is competition, as well as the bandwidth it achieves at high priority during the warmup and measure phases, which is 50%. Once the regular flow finished, there is no competition, as such the flexible at that moment will perform full delivery. For these reasons, at $r < 0.6$ there is no further degradation of the flexible flow. At $r = 0.0$, the flexible flow completes 65%. The fastest rate of the regular flow is similarly limited, to around 7.1 Gbit/s on average, which corresponds to an FCT of 1.12 s – which we approximately observe in Fig. 2.6b with 6.9 Gbit/s and an FCT of 1.17 s.

Another interesting case is for when the regular flow would continue for the entire



(a) *Partial delivery* **(b)** *Flow completion time*

Figure 2.6: Results for a single flexible flow of different reliability factor r competing with a single regular flow. The flexible flow discards data in order to speed-up the regular flow.

duration of the flexible flow. In that case, it would complete at a rate of 2.5 Gbit/s and drain budget at a rate of 2.3 Gbit/s. As such it would finish after 1.66 s with a completion percentage of around 52%, which means that reducing r beyond 0.52 would not yield any speed-up or further partial delivery.

Partial delivery takeaways: REFLEX trades off data loss proportional to its reduced rate over time. Only if there is competition will it complete partially. The amount it can discard is determined by parameter r as well as the probing parameterization: the longer the exploit phase, the more data it can potentially discard.

2.5.4 Challenge of probing

The probing mechanism is parameterized by the interval duration T_{int} and the balance between phases (D_{warmup} : $D_{measure}$: $D_{exploit}$). These probing parameters impact (a) the maximum possible speed-up REFLEX can fundamentally offer, and (b) the ability to fulfill its guarantees.

Interval granularity (T_{int}). We set out to have T_{int} be the minimum interval needed to acquire a measurement of the fair share rate. As such, it is desirable for it to be at least one full RTT (including queueing delay), such that its measurement captures at least a full in-flight congestion window. Likely, several RTTs are required for a consistently accurate fair share estimate. In our simulated network, the RTT over a single link is in the order of several microseconds when empty, and 100s of microseconds when queues are filled. To investigate this granularity, we perform Case 1 from §2.5.2 for three different T_{int}: 1 ms, 2 ms, 5 ms, and 10 ms. For each parameterization, we plot the R_{fair} which was found during the period of competition between approximately the 1.95 s and 2.4 s timestamps in Fig. 2.7. At $T_{int} = 1$ ms, the estimate is considerably lower than the actual fair share rate of half. This is caused by the congestion control protocol taking time to converge. At a T_{int} of 2 ms, 5 ms and 10 ms, the fair share rate is correctly maintained around 50% of the line rate. As a new flow starts exactly at $T = 2\ s$, the initial estimate is a bit higher as the flows take time to balance their congestion window especially with slow start. There is a fine balance between the interval duration and the responsiveness of REFLEX: the lower the interval duration, the quicker a flow can decide if it can become flexible, as well the more recent and thus accurate it is. However, if it fails to converge and obtain an accurate measurement, the fair share estimate will be inaccurate and lead

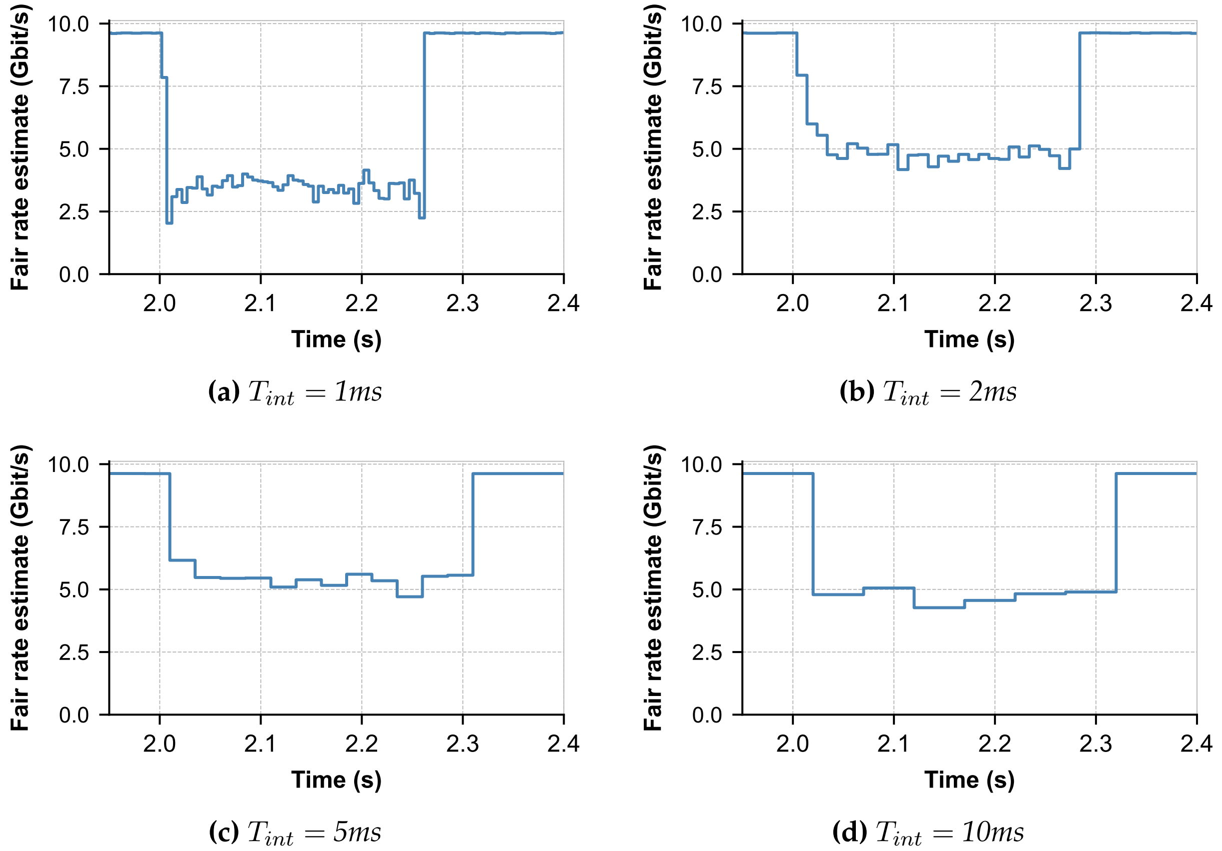


Figure 2.7: Fair share estimated rates for Case 1 for different interval duration. A too low interval duration leads to too low (or incorrect) fair share estimates.

to guarantee violations. As a balance between these two effects, we set the interval duration to $T_{int} = 5$ ms as default. It is important to note that in settings where RTTs are higher, convergence rates will similarly be slower, as such in such networks the interval would need to be higher. Conversely, it could be set lower in networks with even lower latency (and higher throughput).

Phase balance (D_{warmup} : $D_{measure}$: $D_{exploit}$). The ratio between the phases has a profound impact on the speed-up that REFLEX can deliver, as only during the exploit phase can other flows be sped up. We have earlier set the interval duration T_{int} to be high enough to achieve a good enough estimate, and low enough to not cause outdated estimates by itself. With this in consideration, we set both the warmup D_{warmup} and $D_{measure}$ to be 1 in the phase balance. Thus, only the configuration of $D_{exploit}$ remains: the lower, the fresher the fair share estimate is, and the higher, the more speed-up a flexible flow can provide. The speed-up is due to the base proportion of exploit phase

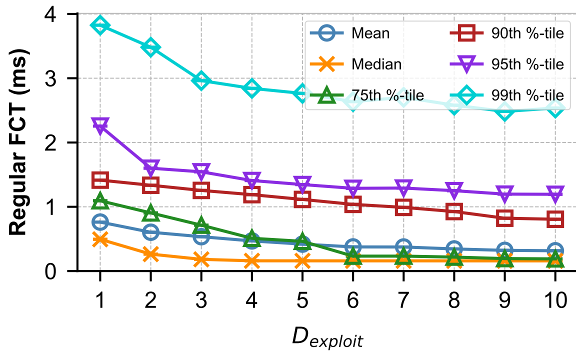


Figure 2.8: Scenario of a single large flexible flow with many short regular flows arriving. The relatively larger the exploit phase is, the higher the percentiles of the short regular flows that are sped up. However, this only holds if the flexible flow has time to accumulate budget first.

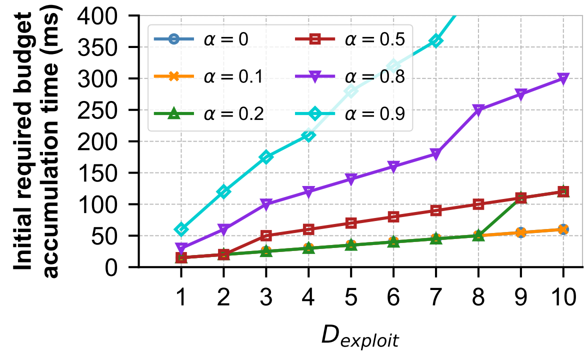
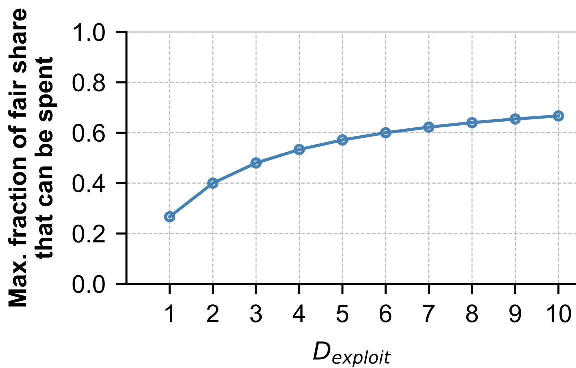
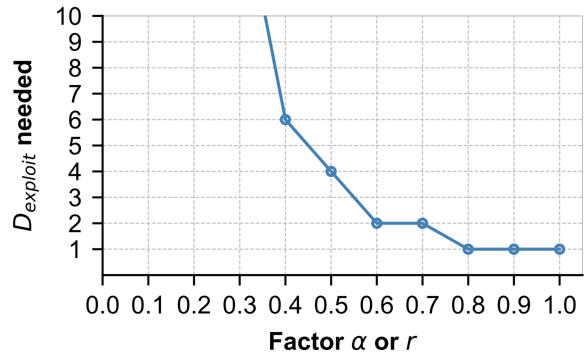


Figure 2.9: Calculation of the required time it takes to collect sufficient budget to go at low priority during the exploit phase with $T_{int} = 5$ ms. At $D_{exploit} = 3$ with $\alpha = 0.9$ it takes 175 ms to collect initial required budget.



(a) *The maximum fraction of the fair share a flexible flow can spent.*



(b) *The exploit phase size needed to be able to spend as much as the factor freedom permits.*

Figure 2.10: Calculation for two flow competing, one regular and one flexible. The longer the exploit phase, the larger portion if it time it can use to speed up other flows. The limits are determined by the time spent in warmup and measure phases (1:1), as well as the underlying priority scheme (which is 9-1 weighted priority in our experiments).

(as is depicted in Fig. 2.10a), as well as the probability of a (short) flow starting and completing wholly in the exploit phase. To showcase the latter, we run 1 long flexible flow with $\alpha = 0.9$ (which is active the entire duration) and short regular flows of 100 kB at a Poisson arrival rate of 1000 flows/s for 5 s with 1 s experiment warmup and 1 s cooldown. We vary $D_{exploit} \in [1, 2, ..., 10]$.

We show in Fig. 2.8 the FCT of the regular flows as we increase the length of the exploit phase. The longer the exploit phase, the better higher percentiles perform. This is as expected: for example at $D_{exploit} = 3$, approximately 60% of the time the flexible flow can operate at low priority as the remainder it spends on (preparing) estimation of the fair share. As such, it is especially effective for the median. A longer exploit phase in general improves even the percentiles which fall out of it, as flows finish quicker thus lessening chance of competition, as well as it reduces the frequency at which the queue size changes suddenly due to the flexible flow switching priorities. However, conversely, it increases how much budget must be accumulated as well as increases the lower flow size bound of flows that can react. We plot this need for accumulation in Fig. 2.9. The exploit phase duration influences the amount of fraction of fair share that can be spent as well, which is shown in Fig. 2.10a and 2.10b. Based on these various tradeoff, we set $D_{exploit}$ to 3 by default. This enables REFLEX in our experiments to spend 48% of its fair share in a two-flow competition (Fig. 2.10a) thus providing benefit for α and r that are greater than or equal to 0.6 (Fig. 2.10b). This parameter value provides a positive impact on the mean, median and the higher percentiles as is shown in Fig. 2.8. REFLEX with $D_{exploit} = 3$ is able to accumulate enough budget for $\alpha = 0.9$ in 175 ms (Fig.2.9). By adding on a cycle (25 ms) for accumulated budget to be spent, we set our flexible flow size in the larger experiments to last 200 ms at perfect line rate (10 Gbit/s) which equates to 250 MB.

The fair share estimate operates at a set granularity, which is based on the convergence rate of the congestion control protocol given the network conditions (in particular, bandwidth and latency). As a first consequence, flows that are potentially flexible but are too short are unable to achieve estimates before finishing. A second consequence is that frequent short flows, which by definition do not have time to fully compete or convergence before completing, will proportionally reduce the fair share estimated by a long flexible flow. This is demonstrated most clearly in Fig. 2.11, which plots for the $D_{exploit} = 3$ the achieved rate and the fair share estimate – in the periods where it is low priority, it still experiences the fair share estimate, as such not resulting in budget reduction

Convergence dependency. We show the convergence behavior of REFLEX by starting a long flexible flow every 200 ms for 4 seconds. The REFLEX convergence ability is dependent on ability of the underlying transport protocol (in this case, DCTCP) to converge when changing queue. We observe the achieved rates as well as their

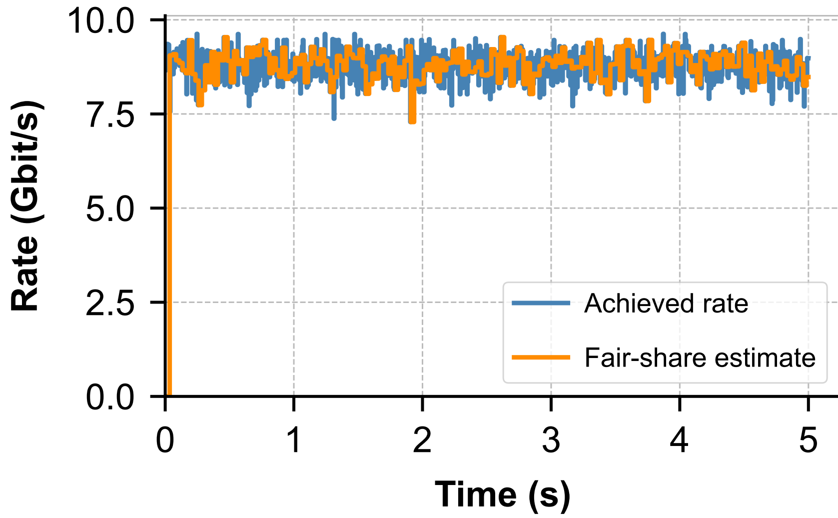
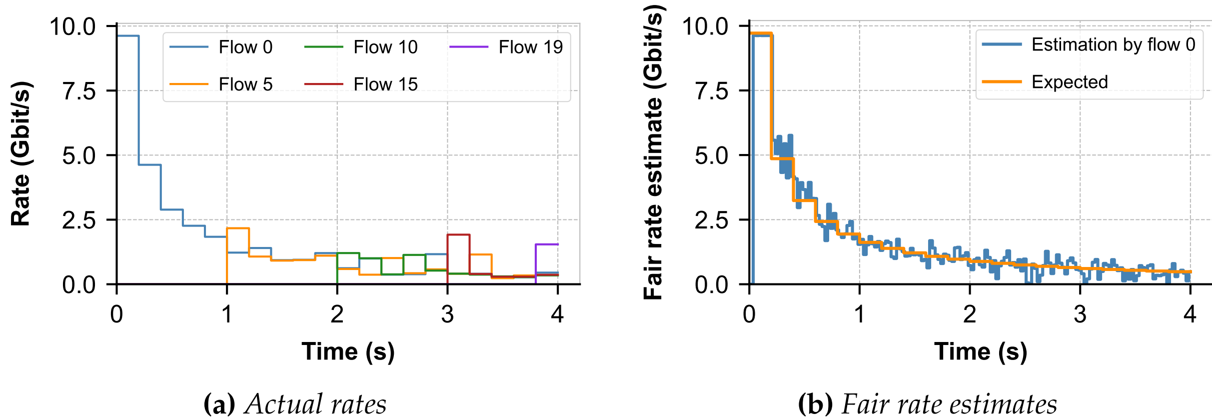


Figure 2.11: Achieved rate and fair share estimate for $D_{exploit} = 3$. Due to the arrival of many short flows, the achieved rate at low priority is the same as at high priority, and as such the estimate is as well.



(a) *Actual rates*

(b) *Fair rate estimates*

Figure 2.12: Actual rate and fair share estimate for the scenario over a single link in which flexible flows arrive every 200 ms for 4 seconds. As the number of flows increases, the variation of the fair rate estimate increases.

estimation of the fair rate. In the presence of many flows, arriving flows are able to achieve a rate similar to their fair share rate. However, at larger number of flows (in particular, beyond 10 flows at $T = 2$ s), there is increased variance and fairness is more difficult to achieve (depicted in Fig. 2.12a). This also reflects in the fair share rate estimate, which exhibits similarly exhibits variation. Although the estimate hovers around the expectation (see Fig. 2.12b), this will result in excessive draining (in case of a too high estimate) or build-up (in case of a too low estimate) of the budget. The ability of REFLEX

to upholds its guarantees is tied to the ability to estimate fair share and thus the ability to quickly converge.

Probing parameterization takeaways: The interval duration must be long enough for the flows to converge and achieve an accurate fair share estimate. This depends on both the throughput as well as latency of the network. The finer grained the interval duration and the shorter the exploit phase, the less time is required for a flexible flow to accumulate enough budget to go at low priority. The longer the exploit phase, the more budget can be spent and as such the more regular flows can be sped up, however the less recent its fair share estimate is.

2.5.5 Utility of increased flexibility

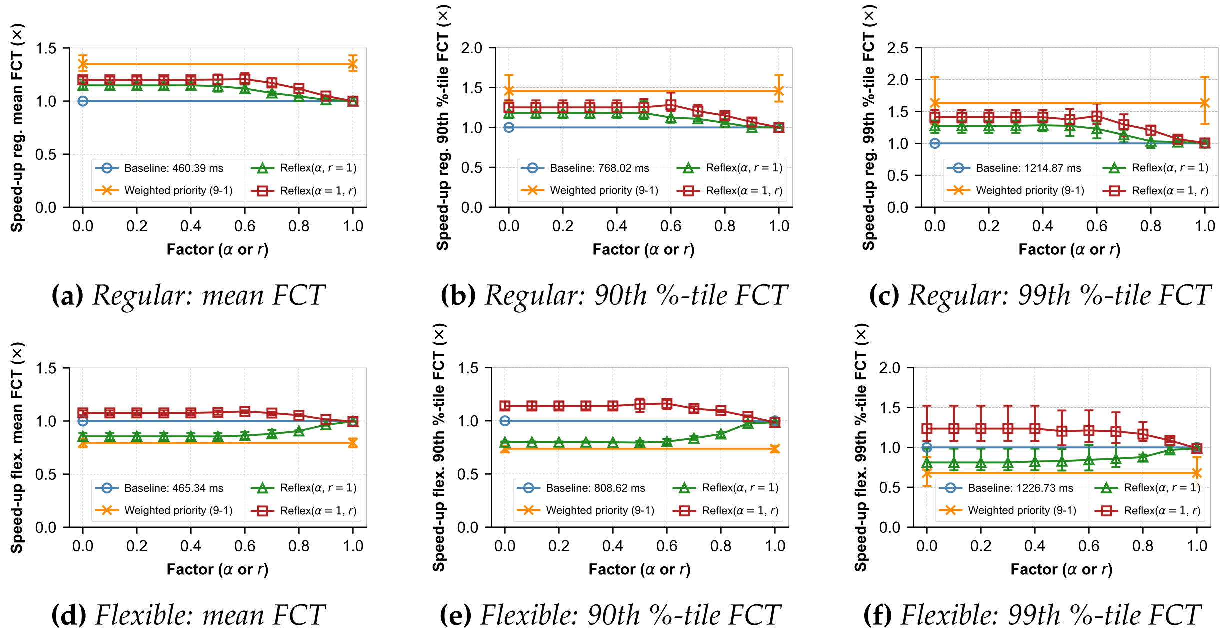
Beyond the configuration of probing, the potential improvement in regular flow performance that is achievable with REFLEX is determined by its parameters, namely α and r. We consider a single ToR network with 20 servers underneath, connected by 10 Gbit/s links. The large flexible flows are set to 250 MB, which is the smallest flow size for the chosen REFLEX parameterization for flexibility to provide benefit, as explained in §2.5.4. This size is both considered a large flow for data centers [6,7] as well as in the order of magnitude of medium-sized ML models such as ResNet [84,212]. As [8], we make use of Poisson arrival of flows which achieves a target average utilization while also providing varying low and high utilization of the network (*e.g.,* also described as microbursts [217]). In the workloads, we aim for 40% average utilization, which is higher than the 25% indicated by [182] as yielding increased drop rate. Half of the target utilization we set to come from regular flows, and the other half from flexible flows. It is thus representing a period of higher than usual utilization in which there is significant probability of flow competition. The 250 MB flexible flows arrive at an average Poisson arrival rate of 20 flows per second, thus providing on average utilization of 20%.

The experiment is run for 16 s (1 s warm-up, 10 s measure, 5 s cool-down) and is repeated three times with different random seeds for its workload generation. In the figures the mean values are plotted with errors bars representing minimum and maximum across the three repetitions. We run REFLEX with full reliability ($r = 1$) but a varying aggressiveness factor $\alpha \in [0, 0.1, ..., 1]$, as well as with reduced reliability ($r \in [0, 0.1, ..., 1]$) but fair share aggressiveness ($\alpha = 1$). For comparison, we additionally run the baseline

strategy without prioritization and the fixed 9-1 weighted prioritization. We consider three workloads for the regular flows.

Workload 1: large flow competition. In this first workload, the regular flows are similarly large (250 MB) and arrive at the same Poisson arrival rate of 20 flows per second. This achieves a utilization of around 40%. The resulting mean and 90/99th %-tile FCT for both regular and flexible flows are shown in Fig. 2.13. Without prioritization, no flows receive special priority, as such the mean FCT of regular flows (460 ms) and flexible flows (465 ms) as well as the 99th %-tile (respectively 1215 ms and 1227 ms) are similar. With the 9-1 weighted prioritization, flexible flows are degraded significantly (speed-up mean: 0.80×, 99th: 0.67×) to accelerate the regular flows (mean: 1.35×, 99th: 1.64×). REFLEX due to the probing phases is unable to provide as much speed-up as weighted priority. However, with both reduced reliability and reduced aggressiveness yields a speed-up for regular flows. At α or r less than 0.5, REFLEX does not yield much additional speed-up as with the network load it is able to consistently be at low priority during exploitation phases. This is in line with prior parameterization observations in §2.5.4. At $\alpha = 0.8$, REFLEX achieves a mean FCT of 513 ms for flexible flows (speed-up: 0.91×) and 441 ms for regular flows (speed-up: 1.04×), with 99th %-tiles of 1390 ms and 1178 ms respectively (speed-ups: 0.88× and 1.04×). In terms of achievable speed-up, the 9-1 weighted prioritization yields significantly more which comes at the cost of being worse for the flexible flows.

We next investigate the ability of REFLEX to uphold its guarantees by using the FCT for each flow in the baseline without prioritization. We observe how much the effect of going at low priority either due to REFLEX or due to weighted prioritization. We plot the CDF of the FCT and speed-up for both REFLEX at $\alpha = 0.8$ and 9-1 weighted prioritization for workload 1 in Fig. 2.14. As noted earlier in §2.2.2, in multi-hop networks a portion of flows will inevitably experience slowdown even if they remain at high priority. This is shown in Fig. 2.14a, where a small tail of around 24% experience a worse FCT for the fixed weighted prioritization scheme – a tail is observed for REFLEX as well of 45%. Weighted prioritization causes a portion of the flexible flows to be starved, which is shown by the long tail in Fig. 2.14d. In contrast, REFLEX does not exhibit such a long tail, which however does go at the cost of significantly less speed-up of regular flows as is shown in Fig. 2.14a. REFLEX guarantees are regarding the fair share rate, not achieving a particular FCT or rate. Nevertheless, it is interesting as a proxy to observe how many of the flexible flows experience a speed-up of less than 0.8 in Fig. 2.14b: around 14.5% has



(a) *Regular: mean FCT* **(b)** *Regular: 90th %-tile FCT* **(c)** *Regular: 99th %-tile FCT*

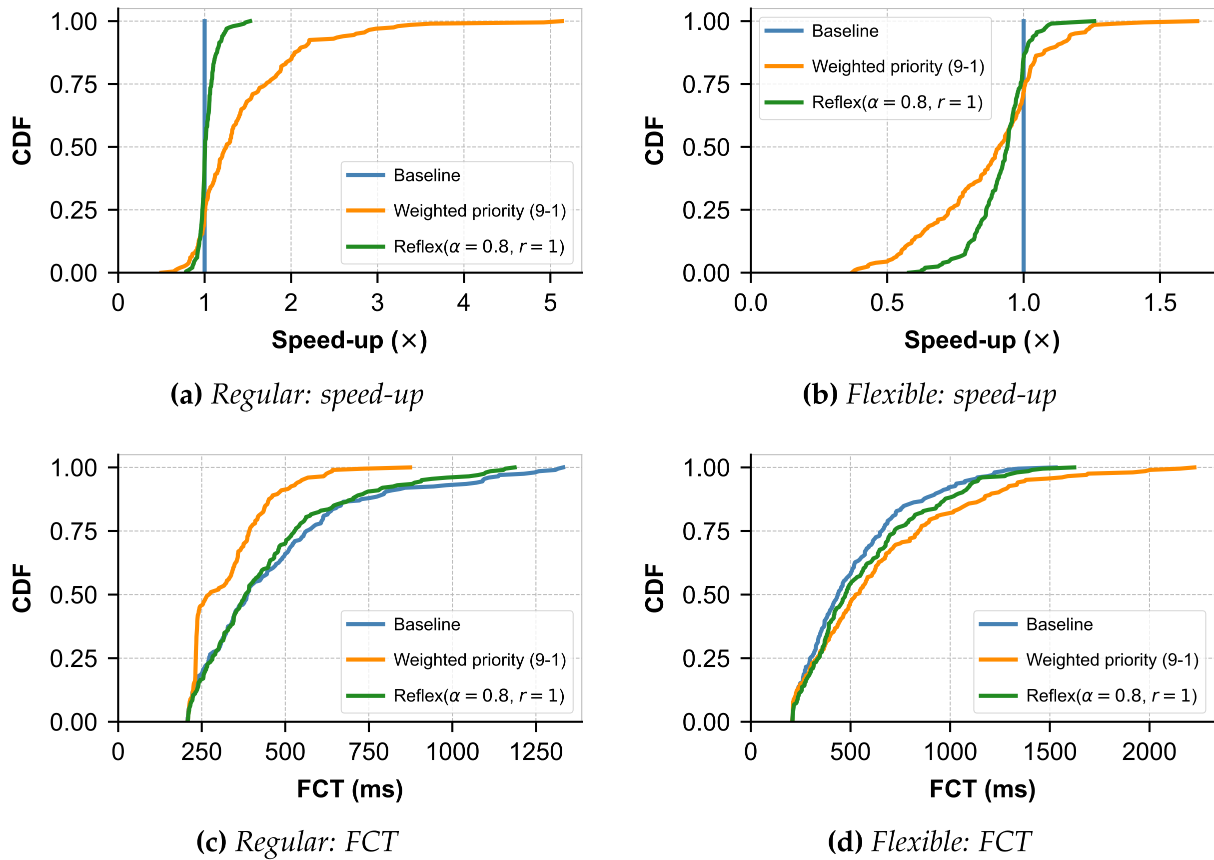
(d) *Flexible: mean FCT* **(e)** *Flexible: 90th %-tile FCT* **(f)** *Flexible: 99th %-tile FCT*

Figure 2.13: Workload 1: large flow competition. Speed-up for the mean and 90th/99th %-tile (higher speed-up is better). REFLEX provides a tradeoff between no and full prioritization, configurable by α and r.

a speed-up less than 0.8. In contrast, the fixed weighted prioritization has 32.5% with a speed-up less than 0.8. The cause why REFLEX is unable to uphold a speed-up of 0.8 is (a) fundamental, as the fair share is interdependent as such a flow going high does not guarantee an R_{actual} greater than the fair share, and (b) practical due to inaccuracies in measurements and the dependency on fast convergence.

> **Takeaways from workloads with large regular flows:** For large flows competing, REFLEX provides a tradeoff between prioritizing the flexible flows and regular flows. Due to its efforts to maintain the guarantees of flexible flows, it provides less speed-up than the fixed weighted prioritization scheme. Its ability to enforce guarantees is influenced by the flow fair share interdependence, measurement inaccuracies and dependency on convergence.

Workload 2: many small flows. In this second workload, we run small regular flows of 100 kB at an Poisson arrival rate of 5000 flows/s. This similarly achieves a utilization of around 40%. This setup mimics a scenario where an α-flexible workload sending a small number of large messages is colocated with a latency-sensitive workload with a large number of short messages, *e.g.*, ML training colocated with Web search. The resulting



(a) *Regular: speed-up*

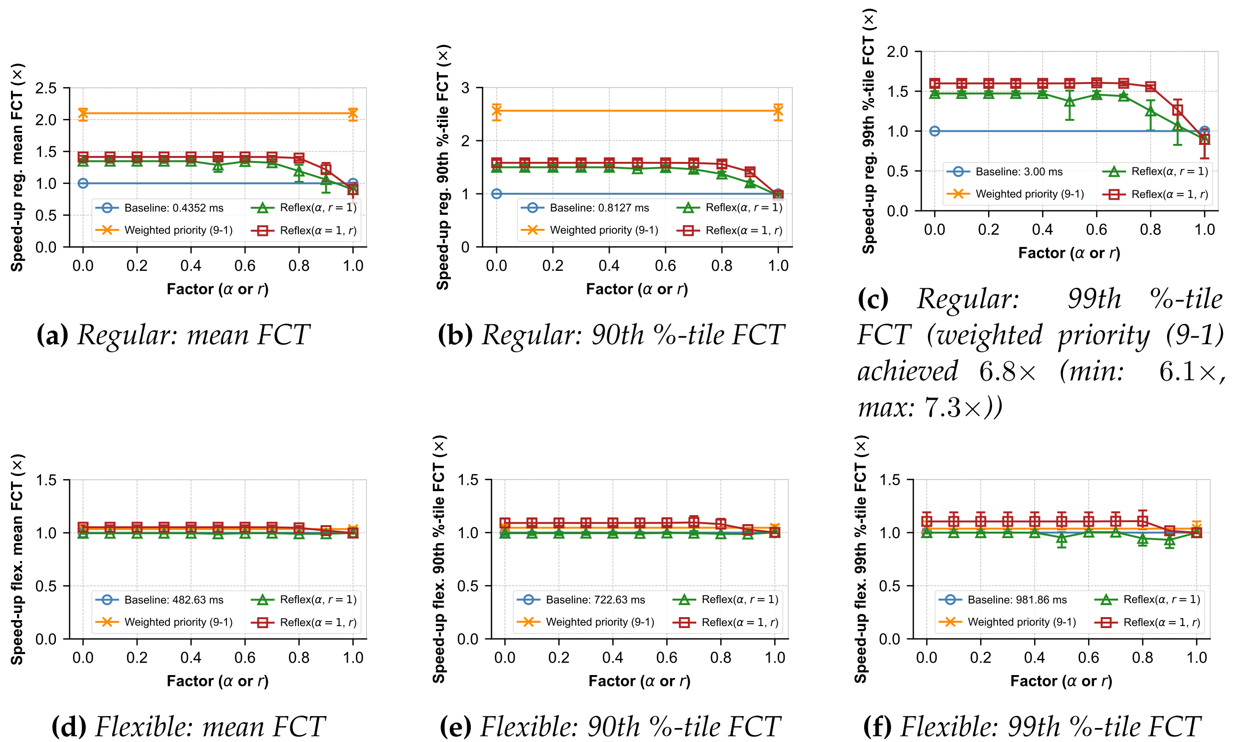
(b) *Flexible: speed-up*

(c) *Regular: FCT*

(d) *Flexible: FCT*

Figure 2.14: Workload 1: large flow competition. For one repetition, the distribution of FCT and corresponding speed-up relative to baseline of equal priority. REFLEX limits the slow-down of flexible flows as it strives to uphold its guarantees at the cost of achieving less speed-up for regular flows.

mean and 99th %-tile FCT for both regular and flexible flows are shown in Fig. 2.15. The 9-1 fixed weighted prioritization achieves a large speed-up in both the mean FCT (2.10×) and the 99th FCT (6.82×). Even for the flexible flows a speed-up is achieved in both the mean and 99th of around 1.04×. By separating the short flows from the large flows, queueing is significantly improved, and both performed better. For REFLEX, the probing has significant impact on the speed-up, as queues are particularly slowing for small flows. As a consequence, REFLEX similarly achieves a lesser speed-up, for example at $\alpha = 0.8$ a speedup of 1.19× in the mean and 1.25× in the 99th %-tile is achieved. In the 99th %-tile of the short regular flows (Fig. 2.15c), REFLEX experiences significant amount of variance likely caused by the probing's shifting of queues – even in some cases performing worse than the baseline. The flexible flows are affected with a mean speed-up of 0.99× in the mean and 0.95× in the 99th percentile. For fixed prioritization,



(a) *Regular: mean FCT*

(b) *Regular: 90th %-tile FCT*

(c) *Regular: 99th %-tile FCT (weighted priority (9-1) achieved 6.8× (min: 6.1×, max: 7.3×))*

(d) *Flexible: mean FCT*

(e) *Flexible: 90th %-tile FCT*

(f) *Flexible: 99th %-tile FCT*

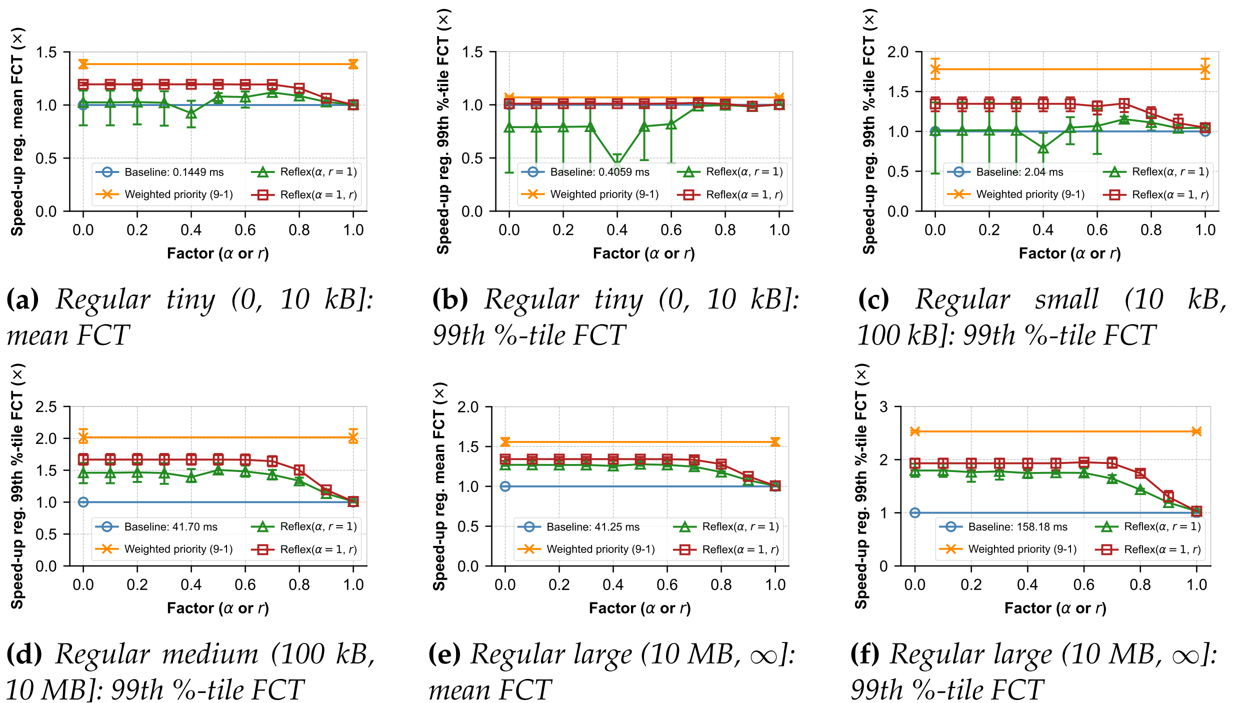
Figure 2.15: Workload 2: many small flows. REFLEX is able to provide speed-up to the short flows, configurable by α and r, however the fixed weighted priority offers significantly more speed-up especially at higher percentiles.

the flexible mean speed-up was 1.04× and 1.04× at the 99th – actually providing a slight speed-up. Our algorithm had 1.2% of flexible flows have a speed-up worse than 0.8×, versus 0.2% for fixed prioritization. Because the many short flows finish in less than the probing phase duration, they become part of the fair share estimate and as such their continuously renewed presence yields little drain of budget. Thus, the point at which reducing α or r further is higher (around 0.7) than for the previous workload (around 0.5). In this workload where many short flows arrive, weighted prioritization yields a better speed-up in FCT for regular flows, while even resulting in a small improvement in flexible flow performance.

Takeaways from workload with many small regular flows: Flexible flows must be of relatively large size, first accrue budget to be able to go at lower priority, and can only be at lower priority for limited time due to probing. Short regular flows finish in the order of 10s or 100s of microseconds, ideally completing only a few round-trips before completing. As such, they are heavily impacted when queued behind

large flows or when receiving congestion signals such as reorder, loss and explicit notifications. REFLEX by frequently switching queues both incurs additional reorder, as well as the instantaneous move of large queues. This particularly negatively impacts the highest percentiles, which are especially important for short flows. Fixed prioritization schemes do not have these effects as flows do not change queue over their lifetime, and as such perform better for this workload.

Workload 3: WS flows. As a continuation of the second workload, we now use the Web Search flow size distribution of pFabric [8] and DCTCP [6] which consists of flow sizes varying between 4 kB and 30 MB, with a majority of its flows under 100 kB. The mean flow size is 1.7 MB, as such to target 20% utilization we configure a Poisson arrival rate of 2921 flows/s. Similar to [7], we separate the statistics of regular flows into four categories of tiny (0, 10 kB], small (10 kB, 100 kB], medium (100 kB, 10 MB], and large (10 MB, ∞]. The results are shown in Fig. 2.16 and Fig. 2.17. We use the 99th percentile as key metric for the tiny and small flows. REFLEX performs especially not well for the tiny and small flows (Fig. 2.16b and Fig. 2.16c), in which it experiences both slower FCT and large variance at lower α. For medium and large flows it does yield speed-up although not as much as fixed weighted prioritization (Fig. 2.16d, Fig. 2.16e, Fig. 2.16f). Flexible flows



(a) *Regular tiny (0, 10 kB]: mean FCT*

(b) *Regular tiny (0, 10 kB]: 99th %-tile FCT*

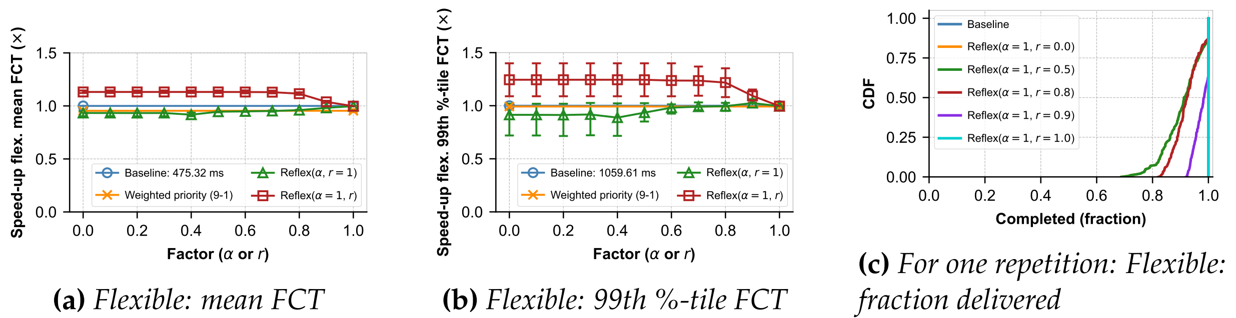
(c) *Regular small (10 kB, 100 kB]: 99th %-tile FCT*

(d) *Regular medium (100 kB, 10 MB]: 99th %-tile FCT*

(e) *Regular large (10 MB, ∞]: mean FCT*

(f) *Regular large (10 MB, ∞]: 99th %-tile FCT*

Figure 2.16: Workload 3: regular flows.



(a) *Flexible: mean FCT* **(b)** *Flexible: 99th %-tile FCT* **(c)** *For one repetition: Flexible: fraction delivered*

Figure 2.17: Workload 3: flexible flows.

are slowed down to bring this speed-up to the regular flows (Fig. 2.17a and Fig. 2.17b). Across all flow size categories, reduced reliability performs well as its reduced delivery reduces general utilization of the network. The fraction of the flows actually delivered before being completed does not exceed the reliability factor r (by design), and lowering the r beyond 0.6 does not yield further reduced delivery as it cannot drain the budget due to the phase balance of 1:1:3 (as explained in §2.5.3). At $\alpha = 0.8$, REFLEX had 1.7% of flexible flows experience a speed-up worse than 0.8× versus 2.2% for fixed prioritization. For the same configuration, REFLEX achieved across all regular flow size categories significantly less speed-up as well, with 1.00×, 1.11×, 1.33× and 1.44× (REFLEX) versus 1.07×, 1.78×, 2.02× and 2.53× (weighted prioritization) for the 99th %-tile FCT speed-up of regular tiny, small, medium and large flows respectively. For Workload 3 as for Workload 2, the fixed prioritization scheme provided better speed-up across the regular flows at less slow-down of the flexible flows.

Takeaways from WS workload: Confronted by a mix of small and large regular flows, REFLEX mostly provides benefit to the relatively large regular flows. The frequent changing of queues of REFLEX combined with the presence of the other larger regular flows led to worse flow completion times of small flows with increased variance at low alpha values. REFLEX performed less well than the fixed prioritization scheme for such workloads in those regular flow size categories while not providing better guarantees for the flexible flows.

2.6 Discussion & future work

REFLEX exposes degradation primitives to applications, thus substantially changing an application-network interface that has remained largely stable for three decades. This new approach raises several threads for further investigation.

Customizing transport: We attempted to minimally change the current network stack to add support for bounded degradation. As a result, our approach inherits the limitations of today's transport. TCP (and its derivatives) are well known to not always converge to fair-share bandwidth, exhibiting various types of divergences from it, *e.g.,* for short flows and flows with different network round-trip times. In practice, our bounded deprioritization guarantee thus is not strict versus the actual fair-share of a flow, but only with respect to what today's transport could achieve. There would thus be value in considering what a clean-slate approach to supporting bounded degradation might look like, and then seeking a suitable pragmatic middle ground between that and our minimal changes. For instance, in a data center setting, it may even be possible to estimate fair-share bandwidth quickly [101, 180, 221]. This would greatly enhance our bounded deprioritization's performance.

Incentivizing applications: Clearly, unless there are appropriate incentives, applications will not ask for degraded service. The simplest incentive in a cloud setting might be cost, *i.e.,* offering slightly cheaper network data transmission for applications in proportion to their accepted service degradation. There is precedent for similar price differentiation for cloud storage offerings with different service guarantees, *e.g.,* for long-term storage [26].

Malicious behavior: Malicious applications could potentially degrade the performance of flexible traffic. Consider an application that splits its large regular flows into many small flows. This can distort the fair-share estimate for flexible traffic, leading it to send at a lower rate. However, this is a problem that exists in today's network stack already: it represents a fundamental limitation of flow fairness, as Bob Briscoe argued [36]. Potential solutions are the same as those that apply to today's transport: instead of flow-fairness enforced at the granularity of transport flows, impose application-level fair sharing using network-level mechanisms.

Coflow support: REFLEX's primitives apply to individual flows. However, for many applications a "coflow" abstraction of their network traffic is more appropriate, whereby

their performance depends on a collection of flows finishing [42]. Extending our primitives to such coflows could potentially increase their benefit, as the degradations could be used more flexibly across large flow collectives instead of in a more restricted manner at a flow granularity. However, extending REFLEX in this manner will require substantial effort, as it implies that the coflow traffic coflow-wide performance rather than local tracking at each flow. Exploring this is thus left to future work.

Other dimensions of service degradation: While we have only explored bounded loss and bounded deprioritization as yet, there may be additional dimensions of degradation worth investigation. One such example is nearly-in-order packet delivery. Instead of TCP's fully ordered delivery, are there ways of benefiting from most data being delivered in order? What's the right framing for a bound on such nearly ordered delivery, and how might be exploit it to benefit other traffic? There is likely some potential in this due to the multipath nature of data center topologies: if some traffic can tolerate nearly-ordered delivery, this traffic could potentially be used for load balancing by suitably packet spraying it, while traffic that needs fully ordered delivery still uses traditional flow-affinity primitives like ECMP-per-flow and can incur link utilization imbalance. However, given the availability of flowlet switching [196], it is unclear how much additional value adding such a primitive brings.

2.7 Related work

In our discussion on the need for new primitives (§2.1), we already contrast our work against standard transport protocols like TCP, UDP, and LEDBAT, as well as work on prioritization and deadline-awareness. Besides these primitives, there is also rich literature on differentiated quality of service [25, 169, 199], which also does not offer guarantees to degraded traffic.

Parallel work available as an online manuscript [129] describes ATP, which provides partial delivery but with the goal of aggressively finishing lossy-flows *even faster*, drawing on the rationale of approximate computing. Unlike ATP, our work is targeted at workloads that instead of aggressively competing for the network, cede ground to time-critical traffic. ATP has loss-tolerant flows behave aggressively which leads to partial delivery through actual packets being lost. In contrast, our approach determines partial delivery based on rate loss relative to the fair share. Further, ATP's complex design

combines packet spraying, separately configured queue sizes for ATP and non-ATP traffic, and a new rate controller. In contrast, REFLEX exploits existing primitives with small changes to how applications transmit data. ATP also does not address bounded deprioritization.

Recent work [206] explored speeding up ML training tasks at the tail by not retransmitting packets that take a long time to be detected as lost, *e.g.,* after a retransmission timeout. This is different from our approach of actively degrading service quality for certain workloads to benefit more critical traffic.

There is extensive work on adaptive bitrate video streaming [133, 209], whereby the video player adjusts its playback quality to network conditions. Note that in this case, application-level quality degradation is a built-in primitive that is tolerated due to the real-time nature of the application: the only alternative to reducing video quality is to suffer pauses in streaming. The applications we consider do not have such real-time constraints and do not accept application-level quality degradation. Thus, the design goals of REFLEX and its implementation share little in common with adaptive video.

The area of streaming analytics has also generated work studying how to best adapt the streamed data to network bandwidth fluctuations [164, 215]. This work takes available network bandwidth as a given, and attempts to find the most suitable way of degrading the streaming data such that the analytics task consuming it suffers the least possible impairment. However, our goals are different: we seek to allow certain flexible workloads to compete less aggressively for network resources by dropping or deprioritizing their traffic. These different goals lead to different design decisions: the streaming analytics work focuses on minimizing application-level degradation, given bandwidth changes, while REFLEX takes application goals as given in terms of degradation bounds, and adapts network-level behavior.

There is also work on approximate network protocols in different contexts, *e.g.,* SAP [165] allows applications to accept partially damaged network data that would otherwise be thrown out by integrity checks like checksum failure. CoAP [181] is a UDP-extension that allows tunable tolerance to network failures in high error rate environments. While philosophically similar, these efforts share little in common with REFLEX in its use cases, the bounded degradations it allows, and its design to enforce guarantees.

2.8 Summary

We make the case that emerging workloads present opportunities for superior network multiplexing by being tolerant of degraded network service. However, to ensure that such applications still get satisfactory performance instead of being excessively penalized for being tolerant, we should offer primitives for *bounded* degradation. We explore two dimensions of such bounded degradation: guaranteed partial delivery and bounded deprioritization. We show that unlike the no-guarantees degradation provided by traditional networking approaches like unreliable transport and various types of prioritization primitives, it is possible to implement bounded degradation to achieve both: (a) improved performance for traffic with strict network service requirements; and (b) assurances that tolerant traffic meets its specified reduced performance requirements.

However, there are practical considerations on (a) the mechanism to actually achieve benefit, and (b) how to achieve awareness of the baseline against which the deterioration is defined. These practical considerations both limit the maximum performance improvement that can be achieved, as well as the ability to fulfill meaningful guarantees. In our work, the continuous probing, frequent convergence of flows and the switching of queues by flows were those practical considerations which impacted performance and guarantees. The two primitives discussed in this chapter are (1) *workload dependent* because they require flexible flows to be starved to outperform other prioritization schemes, and (2) *objective dependent* as they are useful only if it is desirable in moments of high utilization to not fully prioritize regular flows.

By taking first steps in framing bounded degradation for networking, our work opens up several interesting directions worthy of future exploration, including (a) how faster-converging transport can improve the performance of bounded deprioritization; (b) how one might assess and exploit the stability of the workload distribution to configure probing appropriately; (c) in what other dimensions, *e.g.,* nearly in-order delivery, is bounded degradation potentially useful; and (d) how to incentivize good use of bounded degradation for applications that can tolerate it.

3 CHARACTERIZING AND MITIGATING PROGRAMMABLE IN-NETWORK ADVERSARIES

Computer networks are a key infrastructure upon which systems rely for communication to fulfill their function. The central role network infrastructure plays unfortunately has seen it become the target of malicious actors from a variety of attack vectors. These attacks impact large quantities of thousands of switches and span across many entities including ISPs and enterprises [103, 148, 214], and data centers [124]. Vulnerabilities in network devices are frequently reported [44], in which critical vulnerabilities in both management software [47] and the switch software itself [46] are disclosed. For these reasons, beyond necessary measures such as perimeter defense and hardening devices, operators should examine the case in which an attack does actually manage to fully take over network switches (*e.g.,* [146], see §1.2.2 for the full quote). There is a growing concern especially regarding espionage and denial of service, however there is another type of threat which we should proactively consider in this scenario: one in which an attacker rather than espionage or outright denial of service, instead makes use of the available programmability to surreptitiously degrade performance while preventing or impeding diagnosis.

In this chapter, we frame the notion of UNDERRADAR attacks, in which rather than espionage or complete disruption, the aim is to achieve maximum possible degradation of network service, while preventing quick diagnosis and countermeasures by the network operator. We set out to model the role programmable network devices play within this notion. We examine how it expands the capabilities of attackers on one side

as in particular it increases their targeting ability, and on the other side how it impacts existing network monitoring solutions as well how it can be used to improved their detection and mitigation ability. We formulate and evaluate UNDERRADAR attacks targeted at network-, transport-, and application-level. Unless detected and mitigated, these attacks would result in the degradation of applications. We find that existing network monitoring systems have difficulty detecting such programmable attacks, which stems from two insights. Firstly, only a very small portion of packets need to be targeted, at transport-level for instance handshake packets and at application-level due to interdependence of flows for the success of the entire "coflow" [42]. Secondly, the capabilities which enable programmable switches and routers to monitor the network in detail can similarly be used by an adversary to specifically avoid detection. Techniques such as impeding classification of key packets through encryption and obfuscation, and application-aware network monitoring are especially promising to facilitate detection and mitigation.

Chapter outline. In this chapter, we first outline the attacker and operator model in §3.1, in which we characterize their knowledge and capabilities. Second, we characterize UNDERRADAR attacks on the transport layer in §3.2. Third, we characterize UNDERRADAR attacks which are application-aware and as such are aiming to target traffic specifically critical to application performance in §3.3. Fourth, we briefly describe UNDERRADAR targeting towards the network layer and monitoring system in §3.4. Fifth, we look into the ways to mitigate UNDERRADAR attacks in §3.5. Sixth, we look into the related work in §3.6. Finally, we conclude the chapter with a brief summary in §3.7.

Contributions. Text and figures from the following publication were included in this chapter and its corresponding introduction and conclusion:

- Simon Kassing, Hussain Abbas, Laurent Vanbever, and Ankit Singla. *Order P4-66: Characterizing and mitigating surreptitious programmable network device exploitation.* arXiv:2103.16437, 2021. [105]

Notices concerning differentiation of author contributions are made where applicable. In particular, §3.4 references and briefly describes findings made by Hussain Abbas in his Master's thesis [1] in the context of this project.

3.1 Modeling attackers and operators

We start with characterizing both the attacker and the network operator, which forms the basis for our reasoning and examination of effective defenses. We model the goals, knowledge and capabilities of the attacker in §3.1.1. We continue in §3.1.2 with describing the same aspects for network operators, and describe the tools they have at their disposal to identify and remove misbehaving devices.

3.1.1 Attacker model

The UNDERRADAR attacker wants to cause large and obvious degradation in application performance, while making it difficult for the network operator to identify compromised devices and perform diagnosis. More specifically, it makes use of programmability to limit or tailor its interference to prevent network monitoring systems from detecting and localizing its presence.

Operating environment. Networks at scale consists of hundreds if not thousands of switches, cables and servers [182], over which massive amounts of network traffic is sent at any point in time. Network monitoring systems are tasked with detecting problems in these large scale networks and identify devices which misbehave, such that they can be diagnosed and either fixed or replaced. The attacker would like to prevent the operator from diagnosing that some network devices are misbehaving, leaving open the possibility of external causes (*e.g.,* application layer problems; or problems rooted in external networks) for any performance problems experienced by users, tenants, application owners, etc. This scenario is applicable to both the Internet context of ISPs, as well as data centers or enterprise networks. The former is more difficult, as application and network generally fall under different administrative domains. We make the assumption that the attacker has gained control over one or more programmable devices within a network. As discussed in §1.2.2, attackers often can and do take control of network devices. With the likely future spread of programmable hardware, it is reasonable (and important) to examine the setting where these compromised devices are programmable.

Effect of programmability. Devices which have programmable data planes with P4 [34] or equivalent capability are able to perform simple per packet operations at line rate. These packet operations include matching packets, taking actions based on observed

packet (header) data (including editing, dropping and rerouting, as well as generating a limited amount of new packets), and read/write access to memory for stateful behavior. Although the amount of actions are limited as well as their complexity (*e.g.,* no loops, constant time complexity operations), past work has shown they are sufficient for a broad range of applications, including key-value stores [99], in-network aggregation [172, 173] and network monitoring systems [126]. Unfortunately conversely, these capabilities enable an attacker to target important packets and perform more complicated (*e.g.,* statistical or tailored) types of interference which together make UNDERRADAR attacks potentially more difficult to diagnose than those conducted with traditional compromised devices.

Effect of knowledge. The ability of an attacker is tightly coupled to its knowledge of the network, as it enables them to identify important packets or flows, and perform tailored actions. Certain characteristics are common knowledge, such as the high level operation of TCP flows (see §3.2). However, other knowledge will be specific to the network, including its present devices and the topology they form (see §3.4), its monitoring system (see §3.4), and the applications which use it (see §3.3). Some of these can be inferred by the attacker through the controlled devices, *e.g.,* the topology through routing state, or the network monitoring system by what information is logged or at what rate packet samples are collected, etc. Moreover, beyond the most sophisticated operators, relatively simple and standard monitoring systems tend to be ubiquitous. It imperative to limit the knowledge known about the network as much as possible.

3.1.2 Operator model

The network operator wants to provide the best possible service for applications running in or across their network. They want to detect when the network is not performing as expected, and if so, to identify which devices are responsible, and fix or replace them. Operators with different capabilities may draw from a range of monitoring and diagnosis methods. There is a large body of work on detecting faulty devices and anomalous network behavior, ranging from the simplest sampling and probing techniques to more recent work focused on detecting “gray” failures, where devices remain operational, but at degraded functionality [90, 168]. Detecting even benign gray failures is tricky because signals of such failure, such as packet loss and reduced bandwidth, are ambiguous, and can also result from normal congestion. Detection thus often rests on statistically

separating normal and abnormal behavior by processing large amounts of network traces. We discuss below the categories of potential diagnosis approaches, and what they are specifically useful for in general as well as their potential role in countering UNDERRADAR attacks.

Coarse-grained sampling: Widely deployed monitoring systems are largely primitive. For instance, ISPs commonly depend on sampling packets, typically at the rate of one in thousands [160], using tools like NetFlow [49] and sFlow [156]. These (ubiquitous) tools stand out in providing distribution-level information about network traffic, such as the proportion of IPv4 vs. IPv6, or diurnal utilization patterns. However, such solutions under-represent small flows, and can't track packets through the network, making even simple metrics like drop rate hard to infer. Thus, these are mostly ineffective to detect or localize adverserial devices, albeit they do detect large changes in network traffic.

Targeted mirroring makes use of improvements in hardware (in particular, improved programmability) to inspect and decide whether to mirror for every packet at line rate [219], thus not having to rely on random sampling such as sFlow [156]. Mirroring involves cutting the payload, and sending the header to a central collector for broader analysis. EverFlow [219] mirrors TCP packets with important flags (including setup and teardown) and protocol traffic (BGP, PFC, RDMA), and additionally allows operators to set a debug bit on any packet for it to be mirrored. It is important however that only a portion of traffic is mirrored, as else provisioning the network monitoring system network will become prohibitively expensive. Targeted mirroring is useful to safeguard the traffic it targets. In normal operation, it is effective at detecting a broad range of network issues, including silent drops, black holes, load imbalance, unusually high latency, routing loops and protocol bugs [219] as their negative effects are similarly applied to the smaller portion of mirrored traffic. Targeted mirroring significantly reduces the actions a programmable adversary can take to interfere, as for example interference with TCP handshake packets can be easily diagnosed with Everflow. The active component, which allows operators to set a debug bit on packets to explicitly trace certain traffic, is only able to be used for diagnosis if the in-network programmable adversary is not aware of the marks which indicate targeted mirroring.

Active probing is the continuous frequent sending of a small amount of probing traffic between end hosts in order to detect network problems. Microsoft's Pingmesh [76] is one such system, and is able to successfully measure latency as well as estimate packet drops through latency increases due to TCP probe retransmission. By combining the

information collected for many end hosts pairs, Pingmesh is able to pinpoint if reduced network performance (*e.g.,* higher drop rates, increased latency) is local to a particular rack, pod, data center or across all data centers – exact pinpointing subsequently is performed through manual traceroutes. Other approaches such as source routing [75] would obviate the latter, although requiring advanced knowledge of paths. For adverserial devices to be diagnosed through active probing, they must be unable to distinguish between normal traffic and probes.

Path-aware flow statistics systems collects statistics of all flows in the network while being aware of their paths [23, 168]. These statistics can then be aggregated to pinpoint switches and links that underperform – either in absolute or relatively compared to others. The relative comparison is possible because load is expected to be uniformly spread through mechanisms such as ECMP [23, 168]. Microsoft's 007 [23] specifically focuses on packet retransmissions, for which it triggers traceroutes to identify the path. It maps these path observations to individual links, and applies a voting mechanism with threshold to identify links which are exhibiting an outlying amount of drops. Facebook's FB-mon [168] instead collects TCP state statistics such as cwnd, ssthresh, srtt, retx and select duration. FB-mon compares the distribution of these statistics for a link to the aggregate distribution of links which should be experience similar load. If they differ significantly, a link is marked as faulty. They succeed in differentiating from normal congestion drops, anomalous drop rates as low as 0.05% [23] and 0.1% [168] in their respective test environments. There are also other research proposals promising per-packet or per-flow visibility [65, 126, 184]. We expect these to need similar thresholding as discussed above to distinguish anomalous behavior from normal congestion. Such systems, being explicitly aimed at finding partial or gray failures, are poised to be significantly more effective against UNDERRADAR attackers, as they track all traffic (unlike targeted mirroring or active probing).

Traffic validation explicitly focuses on detecting malicious devices [144, 145], by modeling the expected input-output behavior and queueing in the devices, and verifying that traffic observations fit these models. These approaches are tailored to defend against UNDERRADAR attackers, verifying that the behavior of compromised devices fits within its purview, and explicitly comparing reports. However, it is unclear whether such details can be captured for large modern devices, with high port counts, line rates, and shared-buffers.

These different monitoring systems constitute important tools on expanding the range

of interference that is detected as well as limiting the amount of degradation possible. We expand more in later section §3.5 how they fit into an effective mitigation model. First however, we highlight in §3.2 and §3.3 how the programmability that is at the disposal of UNDERRADAR attackers makes it more challenging for the aforementioned monitoring systems to detect them, as it significantly reduces the amount of interactions required to degrade application performance. In addition, in §3.4 we briefly describe the findings for the scenario in which the compromised devices work in consort or specifically interfere with the path detection ability of monitoring systems in order to cast blame on innocuous devices.

3.2 Transport layer targeting

Transport protocols make use of packets to transmit data between endpoints (the payload) as well as information and metadata regarding the connection and the network in which it operates. Depending on the state in which it finds itself (*e.g.,* setup, congestion state, data transmission progress, teardown), certain packets and their respective content can be more important than others. In this section, we will focus on TCP as our prototypical example, variants of which are the most widely deployed transport, both in the wide area and within data centers. We identify two possible outcomes of interference with TCP:

(a) Immediate error. TCP is designed to provide reliable data transfer which safeguards data integrity. Generating new or editing existing packets with certain fields will result in the data transfer to fail, *e.g.,* handshake sequence number disagreement, causing acknowledgment of data which has not yet been delivered, or sending early termination (FIN) signals. The diagnosis of such interference is straightforward as it will result in application errors which directly point at network service failure. In addition, some of these, such as handshake packet interference, can be detected through targeted mirroring approaches such as EverFlow [219].

(b) Performance degradation. Rather than directly causing failure, interference can alternatively affect the underlying congestion control mechanism. This can happen by enabling, editing or disabling TCP options, such as window scaling and timestamp, or by affecting flow control parameters by changing the advertised receiver window size to be small. Similarly, the signals the congestion control uses can be unnecessarily evoked

by dropping, reordering or delaying packets (for delay-based congestion control), or editing congestion markers such explicit congestion notifications (ECN). The editing of options and parameters, can be diagnosed by comparing network logs to system configurations, and highlighting any mismatches. For instance, EverFlow [219] could be used to detect these types of attacks as it mirrors handshake packets, among others. On the other hand, congestion signal interference such as induced loss, reordering, delays and marking are more difficult to diagnose, as they occur occasionally inside the network as well due to queueing and congestion. End-host monitoring can periodically collect and aggregate flow characteristics such as loss [23], or TCP state such *cwnd* and *ssthresh* [168]. These characteristics are then projected on the on-path network devices. It is then decided if flows going over a certain network device are performing worse than others to a certain degree of statistical significance. In a production environment with many flows and noise, these methods cannot detect a single flow experiencing bad performance over a particular link, but only that of a significant portion of flows. Detection requires finely tuned monitoring and analysis which focuses on detecting outliers as well as distribution changes as a whole.

There has been several past network security works that focus on performance degradation attacks, from both the vantage point of third party senders as well as misbehaving receivers. [121] presents a low-rate attack in which a third party sender causes short bursts of congestion timed exactly during retransmissions. [175] explores how a misbehaving receiver can modify how it acknowledges packets, *e.g.*, forcing the sender to ramp up its sending rate faster by splitting each ACK into multiple ACKs for smaller byte ranges. In-network adversaries can also conduct these attacks with minor modifications: for [121] if on the path, it can directly drop packets rather than depending on approximate timing, and if not it could similarly cause such bursts. For [175], programmable switches can parse TCP headers and generate packets (in this case, acknowledgements) as necessary dependent on parsed values. However, the goals and capabilities of in-network attacker differ, as due to their in-network vantage point in-network attackers have direct access to the target traffic going over it, rather than depending on approximate timing [121] or only affecting senders for which it is the receiver [175]. It is thus important for countermeasures to make packets as indistinguishable as possible from another.

We demonstrate the *performance degradation* interference type by emulating a single TCP connection between two hosts over a virtual P4 switch modeled with BMV2 [154] within

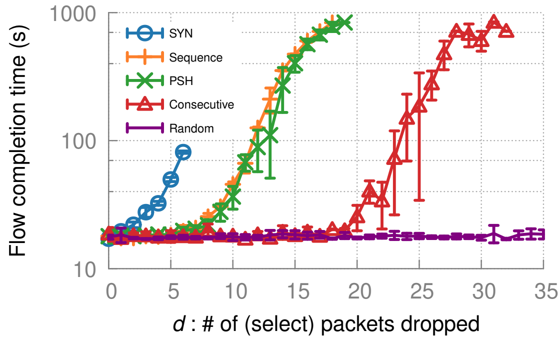


Figure 3.1: Taking away targeting ability prevents slow down of transmission. (Single TCP flow to transmit 10 MB over a 5 Mb/s network in Mininet). (5 repetitions, error bars are sample standard deviation.)

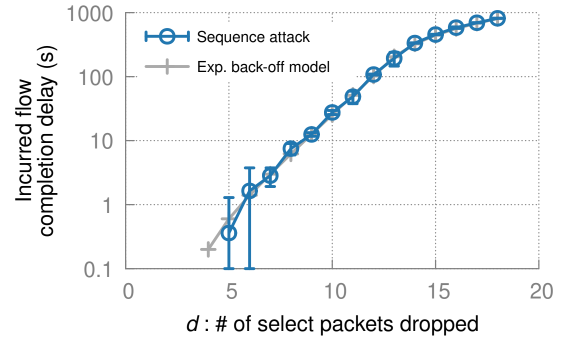


Figure 3.2: The experimentally measured delay from Fig. 3.1 matches the expected exponential back-off. (5 repetitions, error bars are sample standard deviation.)

Mininet [122, 142] based on [155]. The virtual machine has as operating system Ubuntu 16.04.4 LTS with Linux kernel 4.4.0-119-generic. We use the default configuration [127], which has RTO_{min} set to 200 ms, RTO_{max} to 120 s and $RTO_{initial}$ to 1 s. For each run we perform five repetitions. We implement a P4 program for several variants of attacks which repeatedly dropping key packets in a TCP connection (results are depicted in Fig. 3.1 and 3.2):

- SYN drops: The P4 switch consistently drops SYN packets for the target flow. Successive retrieves follow exponential-backoff until the maximum amount of SYN retries (6) is reached, after which the connection timeout.
- Sequence drops: The P4 program arbitrarily selects a sequence number from a target TCP flow, and repeatedly drops packets (retransmissions) with that sequence number (in this experiment the 5000$^{\text{th}}$). Exponential backoff is observed, until the maximum amount of retries (15) is reached, after which the connection times out. In this case, the maximum is reached at 18 drops due to initial packet (1), fast re-transmits (3) and the retries (14).
- PSH drops: There is also a different vector in standard TCP implementations that can be used for an attack similar to the one above storing the sequence number. We observed that retransmitted packets are often marked with the PSH flag. In this experiment after the 5000$^{\text{th}}$ packet, packets with the PSH flag are dropped.

This is a slightly noisier signal than explicitly storing sequence numbers, but is nearly as effective. This requires to store less state per flow.

- Many successive or random drops: For comparison, we also evaluate the effectiveness of two more naive attacks, one where the attacker merely drops random packets, and one where they drop a large number of consecutive packets. Random drops are nearly ineffective. Dropping several consecutive packets (in this experiment, starting at the 2000th) can indeed cause similar slowdown as more informed strategies (as expected), but it requires a larger number of packets to be dropped to achieve this, because a large number of packets are dropped that do not impact retransmission behavior.

Our packet significance analysis as well as the experimental results highlight the importance of impeding the identification of key packets of transport protocols. Moreover, in order for monitoring systems to detect this type of interference, it must operate at fine granularity, match system parameters with observed values in packets, and detect performance outliers as well as distribution shifts as a whole.

3.3 Application-aware targeting

In the same way that the ability to identify key packets must be thwarted (§3.2), the identification of the importance of flows to the underlying application must similarly be prevented. The reason for this is that if the attacker has knowledge of application structure, the hampering of specific target flows rather than randomly can amplify the effect of UNDERRADAR attack, making them application-aware. This prevention is particularly important in settings with fine-grained monitoring, especially in data centers, where it is challenging for an UNDERRADAR attacker to degrade large fractions of flows without being detected by recent, highly sensitive monitoring systems [23, 168].

Many applications in such environments have a large dependency set, and slowing communication between components can slow down the application. This means that network flows are often part of a multi-flow application-level task: a "coflow" [42]. For instance, MapReduce jobs result in data shuffles whereby the map tasks communicate with the reduce tasks, and the job awaits the termination of the last reducer to finish or to start the next iteration. Thus, the progress of compute tasks and the freeing of

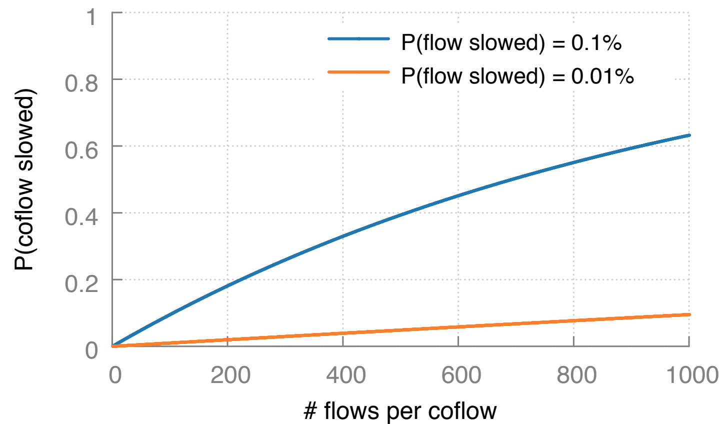


Figure 3.3: In the face of dependencies, a minor probability of a single flow being slowed can yield a major probability of the spanning coflow being slowed.

their resources necessitates that the last communication flows finish quickly. Similarly, user-level responses in Web applications (*e.g.*, search results) often use data requested from hundreds [151] to thousands [95] of components in a "partition aggregate" manner. Given that user-experience in terms of response time is critical to the application providers' revenue and reputation, it is desirable that such coflows (and hence, the last flows within them) finish as quickly as possible.

The completion of such coflows being dependent on the last few straggler flows opens up substantial opportunity for an attacker to degrade performance for a large number of coflows while targeting only a small set of flows. An illustration of this effect is shown in Fig. 3.3. Even if the attacker is constrained by advanced monitoring systems to only cause a slowdown for less than 0.1% of flows, if each coflow comprises 1000 flows that depend on the last finishing flow, fully 63% of coflows experience slowdown[1]. Thus, monitoring systems in networks under such attacks may find that only an insignificant number of flows are impacted, a situation largely indistinguishable from normal congestion. But at the application layer, the large impact would be clearly visible. This is likely to cause a wild goose chase into the myriad of *other* possible causes such as application software bugs, misconfiguration, resource contention, or faulty hardware. The problem could be exacerbated if the network operator and the application owner are different parties, as is the case in popular cloud infrastructure providers with their customers.

How much the slowdown of constituent flows slows down coflows depends on applica-

[1]This effect is well-studied in prior work [52] in the context of failures; our contribution here is in exploiting this natural characteristic of such systems from an attacker's perspective.

tion structure. For example, MapReduce has a hard barrier which causes the slowest flow to determine the coflow completion time. On the other hand, a Web page could potentially be partially returned and still be useful enough. As we shall see later, techniques for straggler mitigation, such as request "hedging" [52], can further soften this dependence, and weaken an attacker.

To illustrate the capabilities of an attacker in such a setting, we simplify the context to a minimal example. Nevertheless, such application structures as modeled, are common in use today.

Application model. We will assume we have n uni-directional coflows going through a single switch, *e.g.*, a top-of-rack switch which hosts the server responsible for user-level jobs or queries. The failure of user-level requests ("coflows") is assumed to impact revenue linearly. (This is likely to be a conservative assumption.) Even failures of small fractions (1-5$\%$) of user queries is disastrous: imagine Amazon's customers experiencing a 1% probability of experiencing a timeout (or missing thumbnails, etc.) on pages they want to load.

Each coflow is modeled as a fan out of m requests to different backend servers going over this switch; $m = 500$ is close to the 521 (average) reported by Facebook for building a popular user-facing page [151], but Web search backends can involve m values in thousands [95]. For simplicity, each request is assumed to be a single-packet UDP flow. We vary the number $F = \{1, 2, 5, 10\}$ of the m requests that have to fail for a coflow to fail.

Hedging. With 500 requests and the failure of only one being enough for the coflow to fail, *i.e.*, $m = 500$ and $F = 1$, even a random drop rate of merely 0.01% will cause 4.9% coflows to fail. This is already unacceptably high, so a request replication technique is commonly used for such applications, called "hedging". For each request sent out, we sent out an additional redundant request, so that the replication factor, $r = 2$. Such hedging's use at Google, together with techniques that eliminate most of its overhead, has been detailed in prior work [52]. Hedging is extremely effective in addressing random drops, leading to a nearly 0% coflow failure rate, because both the original query and its replica query must incur a random drop for the query (and the coflow) to fail.

Misclassification model. To be effective, an attacker needs to be able to classify constituent flows into their coflows, and thus track hedged flows. A request packet is

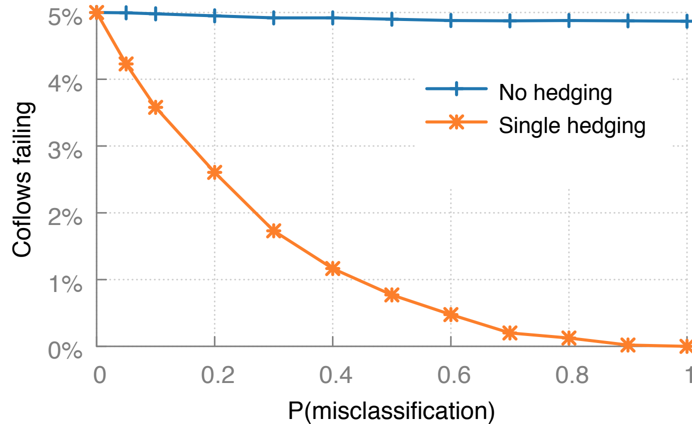


Figure 3.4: Facing imperfect knowledge, hedging reduces the ability to fail coflows ($m = 500$, $F = 1$, $B = 0.01\%$).

identified by the tuple $(id_{coflow} \in \{1, ..., n\}, id_{req} \in \{1, ..., m\})$. The classification ability of the attacker, who has obtained control of the switch, is modeled as a Bernoulli distribution with p_{mc}, the probability of *misclassification*. Thus, the attacker can successfully map a request to its request and coflow identifiers with probability $1 - p_{mc}$. Otherwise, the desired observation of the real (id_{coflow}, id_{req}) is replaced by a uniform random pick.

Budget model. For illustration, we use an overall packet drop rate budget $D = 0.01\%$, which is lower than the detection threshold of state-of-the-art monitoring systems. The total drop budget across all n coflows is then $B = n \cdot r \cdot m \cdot D$ packets. Without hedging the budget is thus $0.05n$ and with hedging $0.1n$. Per coflow, this translates (with $D = 0.01\%$, $m = 500$) to 0.05 or 0.1 packets allowed to be dropped on average. With perfect knowledge about which flows belong to which coflows and which requests are hedged versions of which others (*i.e.*, matching request identifiers), we could fail 5% of the coflows (with or without hedging) by dropping only 1 in 10000 packets ($D = 0.01\%$).

What is the effect of misclassification? We vary the $p_{mc} \in [0, 1]$ with or without hedging, assuming $F = 1$ request must fail to fail the entire coflow (shown in Fig. 3.4). Without hedging, even random classification will cause coflows to fail (4.87%, $\sigma = 0.05\%$) at a nearly equal rate to perfect knowledge (5%, $\sigma = 0$). The small difference stems from multiple requests from the same flow being dropped on occasion, leading to no additional negative impact. With hedging, the reduced ability to classify requests causes significantly fewer coflows to fail. With completely random loss ($p_{mc} = 1$), zero coflows failed in our tests, confirming analysis in previous work [52] showing the effectiveness

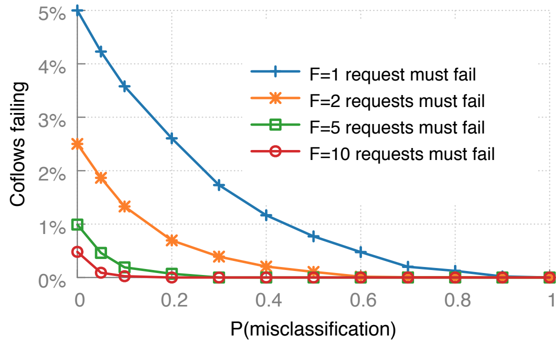


Figure 3.5: The more requests must fail (F) for the coflow to fail, the less coflows fail ($m = 500$, $B = 0.01\%$; single hedging).

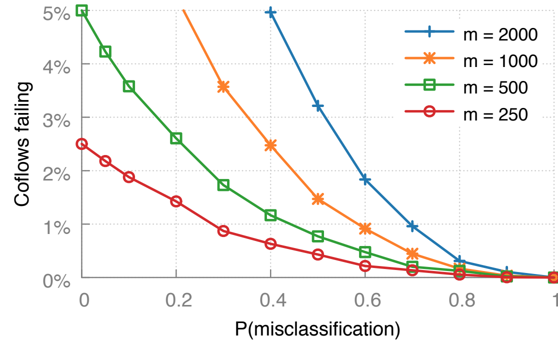


Figure 3.6: The smaller the fan-out (m), the less coflows are able to be failed ($F = 1$, $B = 0.01\%$; single hedging).

of hedging against random loss. However, as the attacker becomes more capable, *i.e.*, p_{mc} decreases, hedging becomes less effective; an attacker with 55% classification accuracy ($p_{mc} = 0.45$) can cause 1% of coflows to fail.

How vulnerable can the coflow be? We vary the $p_{mc} \in [0, 1]$ with hedging, for a range of $F = \{1, 2, 5, 10\}$ requests needing to be failed to fail the entire coflow (shown in Fig. 3.5). The more requests that must fail to fail the coflow, the less coflows will fail. This is caused by the increased budget spent to fail the necessary amount of requests. By increasing F, the amplification effect of the attack is reduced. Because F requests must fail, it is necessary for the attacker to drop at least F hedged requests, which becomes exponentially more difficult as the probability to classify correctly decreases linearly. Typically it is not possible to make the coflow more or less vulnerable, as this is decided by the application's nature.

What is the effect of fan-out? We vary the $p_{mc} \in [0, 1]$ with hedging and $F = 1$, for a range of $m \in \{250, 500, 1000, 2000\}$ requests are sent (shown in Fig. 3.6). With a linear decrease in fan-out, the available budget also decreases linearly. It is important to note that for $m = 250$ in Fig. 3.6 is always higher than $F = 2$ in Fig. 3.5, as decrease in budget is of linear effect, whereas the decreasing of F has an exponential effect as p_{mc} increases. Increasing the fan-out to large numbers but keeping the number of requests needed to fail for the coflow to fail (F) the same will result in merely giving additional budget for the attacker.

What would a practical coflow attack need? Our model of coflow-based applications is

not entirely realistic. First, it assumes there is only coflow related traffic. There can also be non-coflow background traffic, but this can only help an attack focused on coflows, by allowing the use of budget acquired from other traffic on the coflow attack, where this budget is most useful. Second, the attacker is most effective when in control of a device through which most or all requests for a coflow transit; in particular, with hedging, the attacker needs to be able to see both the request and its replica. This is plausible, for example, if the attacker gains control of the top-of-rack switch under which a user-facing server resides that issues the many backend requests. Third, the attacker needs to achieve reasonable accuracy in classifying packets. Past work [216] has shown that heuristics using only the source and destination addresses and port numbers, and the flow start time, can be effectively used to map flows to coflows with over 90% accuracy. Identifying hedged requests is also plausible if packets are unencrypted, allowing at least exact matches sent within a certain small time threshold of each other to be easily identified, *e.g.,* with the use of bloom filters in P4 switches. The attacker could compare both entire packets, as well as only the payloads, in case the replicated queries are sent to different servers, and thus have different headers. In the presence of encryption, the attacker would need to rely on timing to identify hedged requests; this could also work because a common approach is to simply retry after a short, fixed timeout.

3.4 Network layer and monitoring system as target

An additional evaluation was performed in the context of this work which examined more closely how well the existing monitoring systems are able to detect UNDERRADAR attacks and additionally investigates network-layer targeted attacks as well as several additional transport-layer attacks [1]. As this analysis was conducted by co-author Hussain Abbas, it is not directly included in this thesis. Additional information can be found in the respective Master thesis [1] and our publication [105]. For completeness, we briefly describe the key findings. As the network monitoring systems depend on path information provided by interacting with the network (*e.g.,* [23, 168]), it was found that they could be misdirected to wrongly identify non-compromised devices as culprits. The topological location of the compromised device can limit which other devices it can cast blame on. Several additional other transport layer attacks including SYN flood, RST tinker, ACK and drop, ECN tinker, and CWND tinker as well as coordinated incast were experimentally evaluated. It was found that none of the monitoring systems as-is are

able to effectively mitigated the attacks if their inherent analysis ability (in particular, path identification) is interfered with.

3.5 Mitigation

Prevention is certainly better than cure, and to that end, standard best practices for network security are certainly helpful. These include safeguarding against physical intrusion [100], using responsible distribution of access and authorization, replacement of default device credentials with strong, private credentials[2], keeping device software up to date, using a network intrusion detection system [32] etc. However, as is known from past experience, from time to time, such measures will fail, especially in the face of determined adversaries. At least some operators are thus considering the threat of compromised devices even in facilities secured with state-of-the-art security measures (*e.g.,* Microsoft, as noted in §1.2.2). Thus, a defense-in-depth approach is warranted. Based on our analysis of what enables an UNDERRADAR attacker to maximize damage, we frame recommendations for mitigating such attacks.

3.5.1 Impede classification of key packets

The success of UNDERRADAR attacks is reliant on two factors: (a) the existence of packets which are disproportionately valuable; and (b) the ability of new hardware to identify and operate on such packets selectively.

If either of these two factors is absent, the impact of a UNDERRADAR attack reduces drastically. Unfortunately, to some extent, both are fundamentally hard to entirely remove – the first stems from application and transport structures that are hard to change; and the second is also key to extracting the benefits of such devices, such as in-network telemetry for debugging. We discuss in the following, countermeasures that are increasingly limiting for both the attacker and (as an unfortunate side-effect) the operator:

- **Encrypt the payload:** Access to the payload would restrict the application-layer knowledge available to an attacker. For instance, directly identifying redundant or hedged queries by matching payload hashes (§3.3) is no longer possible. The

[2]Sometimes, unfortunately, even industry leading vendors leave no possibility for this [46].

attacker would have to resort to likely more inaccurate inferences based on meta-characteristics (*i.e.,* timing, size), and as we have seen, limiting the ability of an attacker to classify packets in this way, sharply reduces the attack's effectiveness. This method also imposes minimal additional burden on the operator, at least in the data center or enterprise context.

- **Encrypt the transport header:** By encrypting the transport header, selecting packets based on *seqno*, *PSH*, and *SYN* flags is no longer possible (§3.2). Not having access to the source and destination port numbers can also limit identification of coflows (§3.3). However, such changes also limit the operator's view; if the attacker cannot identify retransmissions, the operator may no longer be able to do so either, and thus not be able to characterize losses.

- **Obfuscate timing and size:** Both the workload (*e.g.,* "partition aggregate" of §3.3) and the transport protocol state machine can result in predictable sequences and sizes of packets, both in-flow and across flows. By randomly varying timing (*e.g.,* as proposed by [121] for RTO_{min}) and/or padding of packets, this predictability can be reduced. However, this could potentially increase latency for services.

3.5.2 Application-aware network monitoring

As we have seen, in their default configuration, existing systems are not able to detect UNDERRADAR attacks sufficiently. Pushing these systems for greater visibility, *e.g.,* by reducing the monitoring interval or increasing sensitivity would in fact limit an attacker, but opens up the risk of more benign false positives as well. Ultimately, the network is a noisy environment, with benign transient congestion, often due to microbursts [217], and this noise limits how the extent to which carefully tuned malicious behavior can be distinguished. A possible improvement is for network monitoring systems that depend on path-aware flow statistics [23, 168] to not only consider comparison of distributions to detect faulty links or devices, but also focus on identifying flows which exhibit particularly unusual performance degradation. Additionally, the deployment of strategic targeted packet mirroring [219] can be used to limit the types of possible interference.

The other challenge that current monitoring systems face is if the in-network adversary is aware of their presence. Path discovery for instance (which lets such systems ultimately pinpoint which links are faulty) in many approaches [23, 168, 219] itself relies on packet

information which can be altered by the attacker if it is aware of the network monitoring system. Similarly probes [76] or mirrored traffic [219] can become non-representative of the performance of the real traffic if the adversary actively avoids interference with those. These deficiencies are not because these systems are poorly designed – to the contrary, these are the highest resolution approaches known, in many cases backed by theoretical analysis, with substantial testing and deployment effort behind them. Rather, they are focused on failures, and not as suited for detecting deliberately malicious activity. We believe a promising approach to address the latter problem is to monitor directly the attacker's target: applications.

The UNDERRADAR attacker's goal is to cause large and obvious deterioration in the performance networked applications see. Unfortunately, this does not mean that bringing these problems to light is easy. In most cases, the infrastructure will be run by a provider (ISP or cloud data center) for users and application providers. Today, the latter have only slow, manual means for reporting problems they experience. At least in some settings, it would be possible to develop an API that lets applications automatically report *their* experience of network services to the operator. For instance, an application can raise an alarm to a cloud operator if its flow or coflow completion time changes substantially. This approach can be viewed as an application of the **end-to-end argument** [171] – only the application or tenant knows whether the network is working well.

There are also opportunities for the operator to fix the problem in a blackbox fashion: does tunneling this client's traffic (which could make an attacker's targeting harder, or with ECMP, automatically change paths) improve the metrics reported by the client? Such an approach can, incidentally, also work against gray failures. Exploring this possibility, including its potential benefit and its challenges in terms of implementation and stability, is left to future work.

For ISPs, the problem is easier in some dimensions, and much harder in others. It's easier in the sense that the vulnerable coflow structure is largely absent, or manifests with much smaller and less easily detectable coflows (*e.g.,* different objects fetched from different servers to compose a Web page). But it's harder in the sense that an ISP (with the exception of stub ISPs) typically has no direct interaction with most end users. Although it is somewhat harder due to the absence of host capabilities, the ISP however can focus on deploying more fine grained network monitoring at flow level rather than merely coarse bandwidth statistics across aggregates. Similarly the approach of evaluating multiple paths could be applied to find out if *e.g.,* RTT or rate improves

across flows. Most ideal would be the introduction of a feedback mechanism between end user and ISP such that application performance can be provided, combined with the possibility of source routing by end users to compare performance across different paths.

3.6 Related work

We have already discussed recent work on detecting partial failures in §3.1.2. We thus focus the following discussion on literature related to network attacks and their defenses.

Malicious routers: Prior work (*e.g.,* [144,145]) has analyzed scenarios in which a non-partitioning subset of routers is malicious. This work suggest that each device self report fingerprints of its input-output behavior to the other network devices. These fingerprints are validated to detect inconsistencies. However, this work does not reason about inherent uncertainty in the network (*e.g.,* due to congestion or benign failures). It is also unclear whether the model proposed could be translated to practice for today's high port-count, high line-rate devices.

Network anomaly detection: While there is a large body of work on anomaly detection [32], this chapter largely focuses on the attacks like denial of service, network scans, the spread of viruses and worms, etc., rather than on detecting the performance degradation of legitimate traffic by adversaries who attempt to make their impact as indistinguishable from congestion as possible.

TCP attacks: Several attacks on TCP are well known, such as denial-of-service by targeting TCP retransmissions [121], SYN floods that exhaust TCP connect slots [57], and malicious receivers forcing altering the sender's behavior [175]. Our work differs from these attacks in: (a) its goal of making diagnosis hard by budgeting the attack such that it avoids detection by state-of-the-art monitoring systems; (b) its use of in-network programmable devices; and (c) amplifying the effects using application-layer structure.

Detecting on-path network misbehavior: There is also substantial work on detecting which entity on a network path is misbehaving (*e.g.,* Network Confessional [22]). However, work in this direction also does not consider a careful attacker who drops or modifies packets at a low enough rate to resemble normal congestive losses. Further,

such an attacker, by modifying packet TTLs, can actually blame other on-path entities for packet drops.

3.7 Summary

We bring awareness to the possibility of UNDERRADAR attacks, which attempt to cause application performance degradation while remaining hard to diagnose through the use of programmability. UNDERRADAR attackers tread a delicate balance between causing service degradation and getting their compromised devices detected. These types of attacks aim to minimize their amount of interference as well as tailor it to specific scenarios. We make recommendations towards mitigating these attacks. First and foremost, we highlight the importance of impeding the classification of key packets through the use of encryption and obfuscation, which should strongly diminish targeting ability. Most effective network monitoring systems are those which continuously monitor all real traffic rather than depending on probing traffic or selective mirroring. They should be finely tuned to detect not only distribution shifts which indicate faulty links in their pursuit to find gray failure, but also to identify outliers that could have been the result of targeted interference. In particular, we note the importance of making network monitoring systems application-aware, such that any application performance degradation can be directly diagnosed whether it is related to disproportionally degraded network service.

4 ANALYSIS AND SIMULATION TOOLS FOR DYNAMIC LEO NETWORKS

So far, we have focused on the modeling of computer networks, in both the context of improving performance (Chapter 2) and monitoring to obviate performance degradation in operation (Chapter 3). For the conventional static networks asserted in those chapters, there exists a wide variety of tools available (*e.g.,* [152]). In recent years, there has been an emergent new type of computer networks which are set to see global large-scale deployment [74, 132] for which there does not yet exist as extensive set of tooling: low Earth orbit satellite networks. The networks these satellite form are poised to offer Internet services on a global scale [79]. By operating at low orbit, these widely deployed satellites are able to provide unprecedented coverage, latency and bandwidth [28, 186]. However, unlike their geosynchronous counterpart, low Earth orbit satellites move at large velocity relative to the Earth's surface, and consequently also exhibit dynamic connectivity between one another [30]. The latter is especially impactful for the case of inter-satellite links [30, 79, 116], which form a true network of satellites rather than treating them purely individually as relays between ground stations [81].

Computer network research requires tools which are able to model this highly dynamic network. In particular, the research community needs a tool to better model routing, traffic engineering and congestion control [28]. In this chapter, we describe HYPATIA, an analysis, visualization and packet-level simulation framework which makes use of the network simulator ns-3 [152]. We show through application to three different constellations (namely, SpaceX Starlink [188], Amazon Kuiper [119], and Telesat [190])

the usability and scalability of HYPATIA, and its ability to provide valuable insights into congestion control and traffic engineering in LEO satellite networks. HYPATIA was developed with the aspiration to contribute to increased understanding, improved modeling, better use and easier development of LEO satellite networks.

Chapter outline. In this chapter, we first describe the background and motivation in §4.1. Second, we continue with a description of HYPATIA's architecture in §4.2. Third, we explain the experimental setup in §4.3. Next, in §4.4 we evaluate the value of HYPATIA by demonstrating the utility of both the packet-level simulation and standard flow- and path-level analysis it enables. Finally, we provide a summary in §4.5.

Contributions. Text and figures from the following publication were included in this chapter and its corresponding introduction and conclusion:

- Simon Kassing*, Debopam Bhattacherjee*, André Baptista Águas, Jens Eirik Saethre, and Ankit Singla. *Exploring the "Internet from space" with Hypatia.* ACM IMC, 2020. [106]

In the publication, Debopam Bhattacherjee and I contributed equally and are joint first author. While both Debopam and I are responsible for the engineering and system architecture aspects of Hypatia, I focused more on its usability and scalability, and Debopam was in charge of design decisions and incorporating LEO dynamics and visualizations. André Baptista Águas [3] and Jens Eirik Saethre [170] (Master and Bachelor thesis students supervised by Debopam) performed the initial engineering work and experiments. The experimental results and corresponding figures in this chapter are the result of all our joint collaborative effort. My contributions are in improving the usability and scalability of the simulator, and the experimental evaluation, particularly improving the TCP and UDP scalability assessment, the comparison of loss- and delay-based transport protocols combining latency, bandwidth-delay product and internal state (*e.g.,* cwnd, RTT), the general distribution analysis of endpoint pair paths, and the timestep granularity analysis.

* Shared first author.

4.1 Background and motivation

Recent LEO satellite networks differ significantly from both terrestrial static and geosynchronous satellite networks (*e.g.,* [91, 197]) in various aspects due to which existing simulation tools intended for the latter two are insufficient. The most important distinction is the constantly changing core of the network due to the speed required to maintain low Earth orbit, which is made up out of a very large number of satellites which move in predictable orbital patterns relative to each other [30]. This core is connected internally through the use of inter-satellite laser links (*inter-satellite links*, or *ISL*) [30,79,116], and to surface ground stations through the use of radio communication (*ground-station-to-satellite links*, or *GSL*). These links are formed and dissolved with parties coming in and going out of range [30], as well as their properties change during their lifetime (*e.g.,* latency, bandwidth, loss). Satellites are expected to be able to establish several ISLs simultaneously [30, 79, 116], and ground stations to connect to one or more satellites depending on their antenna type [118]. The constellations are set to consists of thousands of satellites, with potentially as many or more ground stations. In contrast to past LEO satellite networks focused on telephony [92,93], these constellations will offer Internet service on a global scale [132].

We poise that current tools are insufficient for a variety of reasons for the network research community to investigate congestion control, traffic engineering and routing. These are relevant to answering open research questions for contemporary LEO satellite networks in congestion control [30], as well routing spanning multiple constellations [79] and between different domains (*i.e.,* between terrestrial autonomous systems and the dynamic LEO satellite networks) [66, 113]. The closest work to ours is [86], which investigates the Iridium and Teledesic polar orbit LEO constellations using a predecessor of ns-3. Although we could have built upon this work directly, instead we opt to use the actively developed ns-3 in combination with an existing ns-3 satellite mobility module [157], which we complement with GSL and ISL connectivity (§4.2). In addition, [86] does not provide experiments analyzing congestion control or traffic engineering impact, and does not have an integrated visualization component. There is also another set of tools which provide useful visualizations and analyses, but do not provide packet-level simulation: (i) [30] analyses LEO topologies in terms of latency and hop counts, (ii) [79,80] visualizes a recently proposed LEO constellation (Starlink) and gives several selected RTT analyses between ground station pairs, (iii) SaVi [204,205] visualizes the

movement of an individual satellite and its coverage, (iv) NASA GMAT [150] provides models for objects related to space missions and generates relevant visualizations and plots, (v) [38] visualizes the Starlink constellation and shows the orbits of individual satellites, (vi) [56] visualizes the progressive deployment of the Starlink constellation, (vii) [53] estimates the total throughput of LEO constellations and optimizes number ground stations to support that throughput, and (viii) [66] only simulates latency and evaluates possible Internet routing schemes for LEO constellations. What sets us apart from these related works can be summarized as follows: we set out design a tool which supports packet-level simulation over large LEO satellite networks with both ISLs and GSLs, combined with helpful analysis and visualization components.

4.2 Architectural design

We built HYPATIA in order to meet the research needs stipulated in §4.1. By modeling at packet-level granularity instead of flow- or path-level, packet level interactions can be researched. The design of HYPATIA models static link bandwidth and varying latency. ISLs are modeled solely as point-to-point with varying latency with a fixed bandwidth. GSL interfaces are modeled to be able to send to any other GSL interface within range (from satellite to ground station or vice versa) with a limited sending bandwidth, but an unlimited receiving bandwidth. This simplified model does not model link layer concerns such as frequency assignment and medium access control, which would more stricter bound transmission rates. The primary motivation behind this is that this simplified model already encompasses many interesting use cases (as we explore in §4.4). As such, additional engineering effort would have to take place to integrate such low lower level mechanisms to HYPATIA. HYPATIA consists out of three main components. First, static and dynamic state of the satellite network over time (in particular, forwarding state) is calculated (§4.2.1). Second, this generated state is used as input for the packet-level simulation (§4.2.2). Finally, there is the analysis and visualization component (§4.2.3), which takes the dynamic state and/or logs of the packet simulator to produce analyses, plots and visualizations of interest.

4.2.1 State generator

The state generator takes as input the parameters which define the ground surface and satellite constellation, including but not limited to the number of orbits, satellites per orbit, altitude, eccentricity, constants including maximum ISL and GSL length, and ground station locations. It uses these parameters together with algorithmic choices (*e.g.,* routing algorithm, ISL and GSL connectivity constraints) to calculate fixed (*e.g.,* ISL topology, satellite movement descriptors (TLEs [111])) and dynamic state (which varies across time). The dynamic state generator is designed to speed-up later simulation by pre-computing state which can be repeatedly reused across different runs over the same network. As an additional benefit, by having this state pre-computed, it also enables analysis of itself. For geometrical and orbital calculations we make use of a range of Python modules, including *pyephem* [166], *sgp4* [167], *astropy* [24], and *geopy* [64]. We make use of *networkx* [78] for path calculations (in particular, Floyd-Warshall) over a graph constructed with calculated link lengths (both between satellites, and to ground stations). It calculates at a fixed time interval granularity (*e.g.,* 100ms) the routing through the topology of ground stations and satellites, and outputs forwarding state changes (and GSL bandwidth changes, optionally) for each time moment.

4.2.2 Packet-level simulator

The packet-level component is implemented as a module for ns-3 [152], a widely used network simulator. It makes use of an existing satellite mobility ns-3 module developed by Pedro Silva [157] to calculate the satellite positions. These positions are then used to calculate when relevant (*i.e.,* upon transmission of a packet) the distance and thus the latency. The forwarding state changes (and GSL bandwidth changes, optionally) are executed as events in the packet-level simulation. It makes use of the basic-sim toolbox ns-3 module [1, 60, 104], which was significantly extended in the context of this work, to assist in routing and topology abstraction and construction, and running TCP and UDP experiments. The packet-level simulation component outputs various logs regarding applications (*e.g.,* flow completion, ping outcomes), transport protocol internal state (*e.g.,* cwnd, ssthresh) and link utilization.

4.2.3 Analysis and visualization

HYPATIA offers various utilities to analyze, plot and visualize the various outputs of both the state generator and the packet-level simulator. We make use of *gnuplot* [202] for graphs, *cartopy* [134] for simple two-dimensional visualizations, and Cesium [39] for two- and three-dimensional visualizations. *numpy* [83] and *statsmodels* [177] are used for calculations. The analyses and visualizations are showcased in §4.4.

4.3 Experimental setup

The investigation into representative scenarios for LEO satellite networks was primarily conducted by co-author Debopam Bhattacherjee. As such, we only present here the key information relevant to the experiments. Further details concerning the constellations and their parameterization based on public announcements and filings can be found in the publication [106]. We consider the first shell of three satellite constellations: SpaceX Starlink S1 comprises 1584 satellites arranged in 72 orbits of 22 satellites orbiting at a height of 550 km [187], Amazon Kuiper K1 comprises 1156 satellites arranged in 34 orbits of 34 satellites orbiting at a height of 630 km [119], and Telesat T1 comprises of 351 satellites arranged in 27 orbits of 13 satellites orbiting at a height of 1015 km [191]. Of the three constellations, Telesat T1 has the lowest minimum angle of elevation to communicate [191], which is the minimum required angle at which a ground station is able to communicate to an overhead satellite (lower is better). The calculation of the maximum GSL length is based on the cone shaped by the altitude and a flat minimum angle of elevation. This calculation used in the experiments does not take into account Earth's curvature, and as such yields a slightly shorter maximum GSL length for Starlink (0.97×) (because cone radius was directly based on FCC filings values), a longer maximum GSL length for Kuiper (1.12×), and especially a longer maximum GSL length for Telesat (2.09×) compared to when Earth's curvature is taken into account. This affects the experimental results in the following way: the longer maximum GSL length is, the more first and last hop candidates there are, thus yielding potentially lower RTT, less hops and increased path stability. In particular in Fig. 4.4, Telesat would have higher RTT and potentially more path changes. Accounting for Earth's curvature improves the calculation as it ensures that the angle from the ground station does not go below the constellation minimum angle of elevation – which can result in line of

sight through Earth's surface if negative. We note the improvement of the maximum GSL length calculation for future work. The default satellite network ISL topology is a mesh-like structure [79, 116] also named +-Grid [30] for each constellation, which means each satellite has four links, with two to the preceding and succeeding satellite in its own orbit, and two to its positional neighbors in the adjacent orbits. We use shortest path routing with distance as the edge weight, thus having between between each source and destination the path with the lowest latency. The ground station location are set to those of the hundred cities with the largest population across the world.

4.4 Utility evaluation

We conduct several experiments to evaluate the usefulness of HYPATIA across a variety of important network research use cases. We first showcase its ability to tackle the scenarios of interest (*i.e.,* constellations and ground stations) at scale in §4.4.1. We proceed with explaining how the model of packet-level interaction benefits investigation of end-to-end congestion control, routing and traffic engineering in §4.4.2. In §4.4.3 we analyze constellation-wide characteristics which underline the need to model the dynamics of LEO satellite networks. Finally, we briefly describe the possible visualizations in §4.4.4.

4.4.1 Scalability

In order for HYPATIA to be useful for LEO satellite networks, it must be able to scale to be able to support hundreds, if not thousands, of nodes in the topology. The state generator performs routing and path finding algorithms at a time interval. It is able for any given time moment using the orbital elements to calculate the topology the satellites and ground stations form. This enables the state computation to be parallelized by assigning each process a portion of time moments and merging them together to calculate only the changes (which saves storage space as not as many changes happen each time step, see Fig. 4.5a). The packet-level component uses discrete event simulation. In discrete event simulation, the scalability is tied to the amount of events and the time spent to execute events.

To showcase the scalability of HYPATIA, we run an experiment over Kuiper's K1 shell of a random permutation traffic matrix between the 100 most populated cities as the

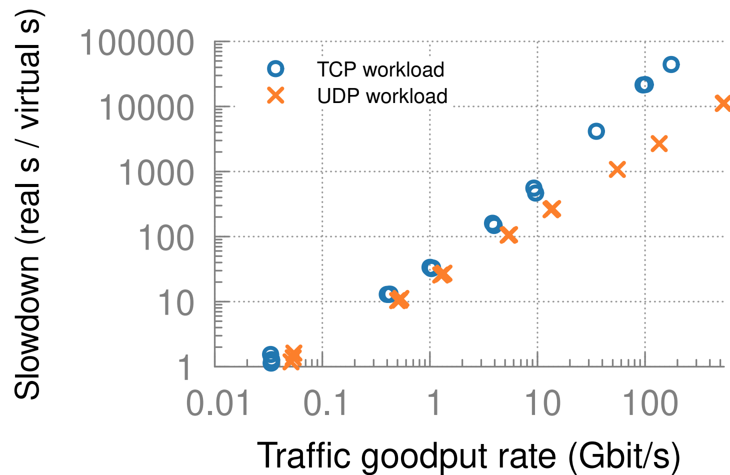


Figure 4.1: Scalability. Running experiments resulting in a 9.2 Gbit/s network-wide goodput with TCP for 1 second takes ~ 555 **seconds. UDP simulations are faster, with 13.8 Gbit/s goodput in** ~ 269 **seconds.**

ground stations. In order to obtain several scaling data points, we vary the link rate of network interface (both ISL and GSL) to change the amount of traffic, testing line rates of 1, 10, 25, 100 and 250 Mbit/s, and 1 and 10 Gbit/s. Between each pair, for the UDP scenario packets are transmitted at a paced constant rate, and for TCP scenario a long running TCP connection. The experiment is run on a single core 2.26 Ghz Intel Xeon L5520. We calculate the rate at which data is acknowledged (TCP) or delivered (UDP) across all pairs. In Fig. 4.1 we plot this rate (*i.e.*, representing amount of activity) against slowdown: the amount of wallclock seconds it took for HYPATIA to simulate a single simulation second. A logical tradeoff is present: for the same amount of wallclock time, one can either choose to simulate a lower traffic rate for longer, or a higher traffic rate for a shorter duration. By extrapolating the discrete points mentioned in the caption of Fig. 4.1, we can approximately estimate that to simulate around 10 Gbit/s network-wide TCP traffic for 1 simulation second takes around 10 minutes, and for UDP traffic around 3.3 minutes.

As traffic is segmented in packets (rather than aggregate, as for flow-level simulation), scalability is tied to (a) the amount of events each packet generates and (b) how many packets are generated. The events generated are both in the network through arrival, forwarding and queueing, as well as at endpoints for transport protocol logic. In contrast to for example data center networks, packets in satellite networks experience more hops and as such generate more events. The difference between TCP and UDP slowdown is caused by TCP warranting additional acknowledgements packets, as well as vastly

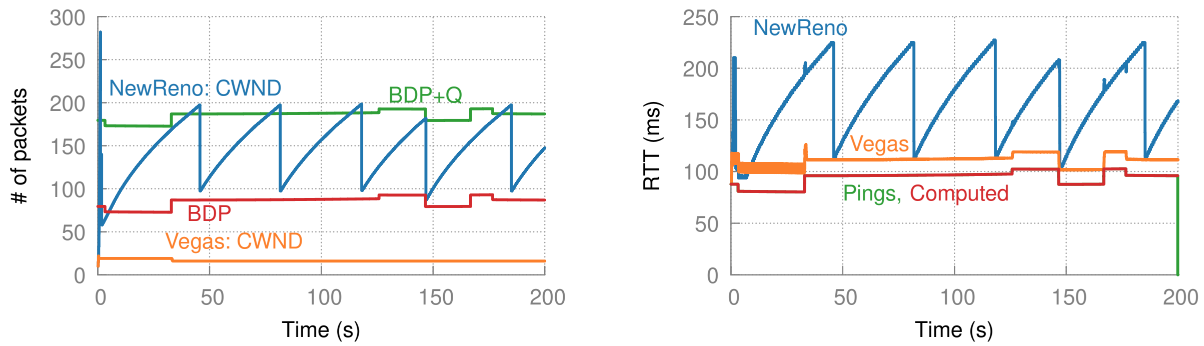
more control logic (*e.g.,* to handle reordering). Although LEO networks are extremely large, inactive nodes (or interfaces) do not increase runtime in discrete event simulation as they do not produce events. As such, besides initial (constant) setup cost, the size of the network on its own is not prohibitive: the activity it generates however, is. On top of ns-3's network device and interface model, HYPATIA adds the calculation of variable latency based on the mobility models on either side, which is a minor overhead. In our design, we set out to pre-calculate and pre-fill as many components as possible, for instance by calculating the forwarding state in advance and pre-filling MAC tables to prevent resolution traffic. The largest potential lies in applying ns-3's distributed mode, which we are currently exploring.

4.4.2 Packet-level interaction

Our key motivation to incorporate packet-level simulation is to observe packet-level interactions, which is not possible through path- or flow-level analysis. We find two scenarios that can benefit from this modeling: end-to-end congestion control and traffic engineering.

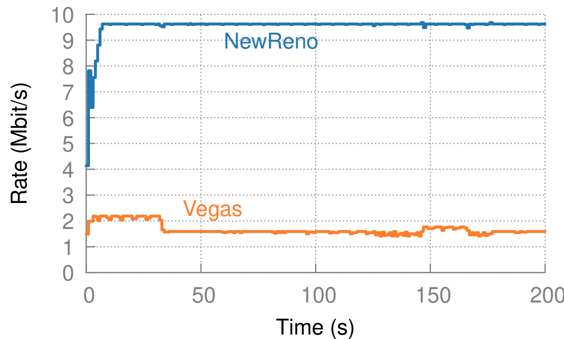
End-to-end congestion control. Transport protocols in particular are affected by packet-level interactions, as they are used as signals to control rate and retransmission. For example, out-of-order packets can be interpreted as loss, and loss or increased delay as congestion. To showcase several interesting phenomena, we highlight one end-to-end communication pair from Rio de Janeiro to London over the Kuiper K1 constellation. We separately run a TCP flow between endpoints and a continuous UDP ping, and complement them with expected latency, RTT and bandwidth-delay product – which are calculated from the fixed and dynamic state files. The queue size is set to 100 packets, which approximately equates to a bandwidth delay produce for 10 Mb/s at 100 ms RTT with a segment size of 1380 byte. The TCP experiment is performed with a loss-based congestion control protocol, NewReno [87], or a delay-based congestion control, Vegas [35].

The resulting figures are shown in Fig. 4.2. NewReno exhibits the familiar sawtooth pattern, which fills the queue until a drop is detected (typically through three duplicate ACKs) and promptly drains it by halving the congestion window. In cases where there is a sudden reduction in latency due to a path change, *e.g.,* at $T = 146.6$ s, this will lead to reordering which results in the same effect. Vegas on the other hand is unable to



(a) *Congestion window, bandwidth delay product and queueing. NewReno exhibits the usual saw-tooth pattern, Vegas is unable to increased due to a too low initial RTT measurement.*

(b) *Round-trip times. The pings closely match the computed RTT via networkx.*



(c) *Throughput. Vegas due to its low congestion window achieves a low rate.*

Figure 4.2: Rio de Janerio to London: end-to-end congestion control with 10 Mbit/s, and queue size of 100 packets which is approximately 1 bandwidth-delay product (BDP) for 10 Mb/s at 100 ms RTT.

sufficiently grow its congestion window, which is caused by an initially low base RTT estimate (due to the small initial handshake packages) which it does not overcome as base RTT is the minimum observed RTT across the entire lifetime. Through the addition of such analysis and plots, we are able to gain insight into factors that congestion control protocols need to take into account. For instance, delay-based congestion control must be aware that the base RTT can vary over time, as such it is not always attainable during their entire lifetime. Another insight is that due to path changes, it is possible for packets to be reordered without it indicating loss, simply due to later packets traversing a faster route. We note that the showcased congestion control protocols are mere examples, and we believe HYPATIA can be of use to more broadly evaluate the performance of existing and novel congestion control protocols in LEO satellite networks.

Traffic engineering. As the end-to-end congestion experiments have indicated, unlike static networks the flows going over LEO satellite networks can experience path, latency and connectivity changes. This is in stark contrast with terrestrial network infrastructure, in which generally the flows during their lifetime are not moved unless by design of network (*e.g.,* to load balance [196]). There is a significant body of work to optimize utilization of terrestrial wide area networks, *e.g.,* Google's B4 [94], Microsoft's SWAN [89] or SMORE [120]. These works cannot be applied directly, as their constraints are vastly different to those required by such dynamic networks as LEO satellite networks. Not only are the links subject to continual latency and throughput change (primarily GSL, but also ISL to lesser extent), but also there are continuous decisions regarding which links should be formed [30]. Moreover, the objectives of LEO satellite network routing might be different than those of terrestrial networks, for instance with additional weight of consideration on path stability, minimization of latency, or maximization of throughput, depending on the use case. [208] for instance proposes incorporation of link state information (*e.g.,* queueing, utilization) into the routing mechanism for LEO satellite networks. Packet-level interaction is vital to understanding how such novel algorithms would perform in these highly dynamic networks, because the continual changing of paths results in the movement of queue packet build-up. This build-up will result in congestion signals such as delay, marking, reordering or loss to the control protocol – its ability to handle these signals quickly and effectively is important. This can include both in-network mechanisms, as well as endpoint congestion control procedures. This level of modeling is not possible with path- or flow-level simulation, which underlines the utility of HYPATIA for this use case.

To provide an indication of these phenomena, we perform an experiment in which we have the pair chosen before, Rio de Janeiro to London, supplemented with a random permutation matching of the other ground stations in the Kuiper K1 constellation. We remove all pairs which have the same destination satellite as our fixed pair, such that the congestion is focused solely on the ISLs of the constellation. We perform a long living uni-directional TCP flow between each pair. We are interested in whether the ISLs of the paths between Rio de Janeiro and London are being used to their full potential, as such we plot the maximum link utilization across the links used in each path over time from Rio de Janeiro to London. We compare this against the case in which we "freeze" the satellite network at $T = 0$, to have a comparison to a static network equivalent. The results are shown in Fig. 4.3. The network of the moving constellation continually

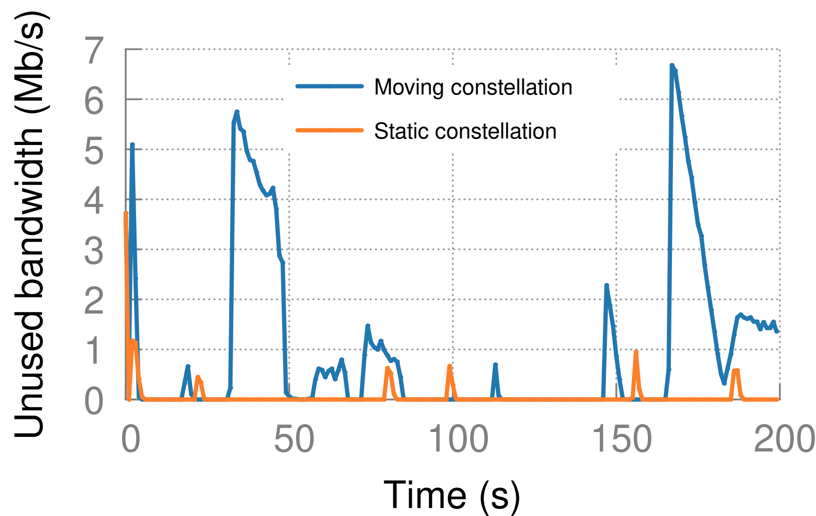
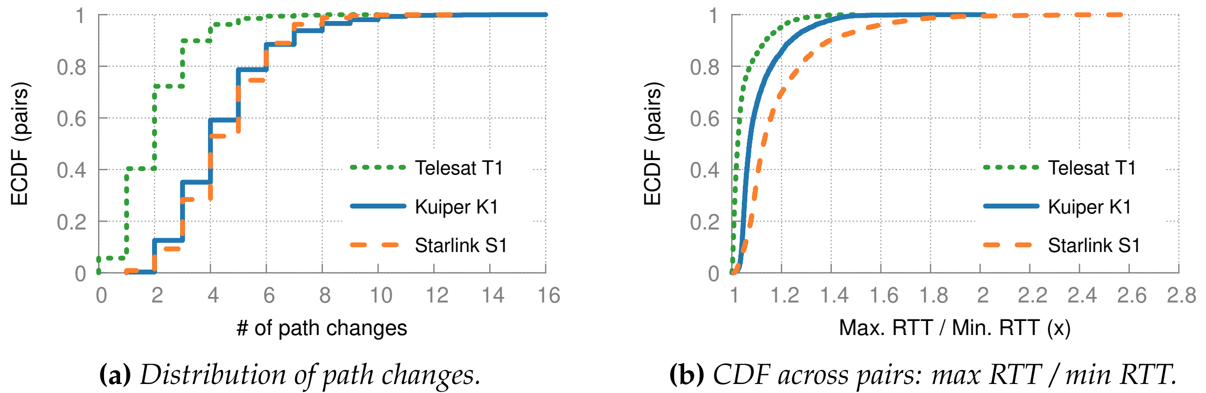


Figure 4.3: Unused bandwidth for path of the same pair as 4.2 with additional cross traffic.

shifts, in which queues are moved, congestion signals are issued, and the endpoints react. As a result, in a portion of time there is a certain amount of lost bandwidth, which is not the case for the static comparison. The cause of this is both due to queueing of acknowledgements behind data packets (which can also occur in static networks), as well as congestion windows not recovering quickly enough due to the long RTT. The congestion window recovery is necessary because congestion signals due to reordering are continuously issued and the steady state changes as flows join and depart links.

4.4.3 General dynamics analysis

Although the primary contribution of HYPATIA is the enabling of packet-level simulation over a moving LEO satellite network, we also pursued path- and flow-level analysis based on the generated static and dynamic state to discover further underlying insights. We are particularly interested in the significance of modeling the effect of dynamics across all endpoints in order to underline the utility of HYPATIA over static network packet-level simulation. Our first point of interest is the stability and latency of the paths between the endpoints. We use the world's 100 most populated cities as ground stations, and between each possible endpoint pair combination determine the path over the course of 200 seconds. For each pair, we calculate the amount path of changes (Fig. 4.4a), and the ratio between the fastest and slowest RTT (path length) (Fig. 4.4b). We find that at least for an algorithm which optimizes for shortest latency, the vast majority

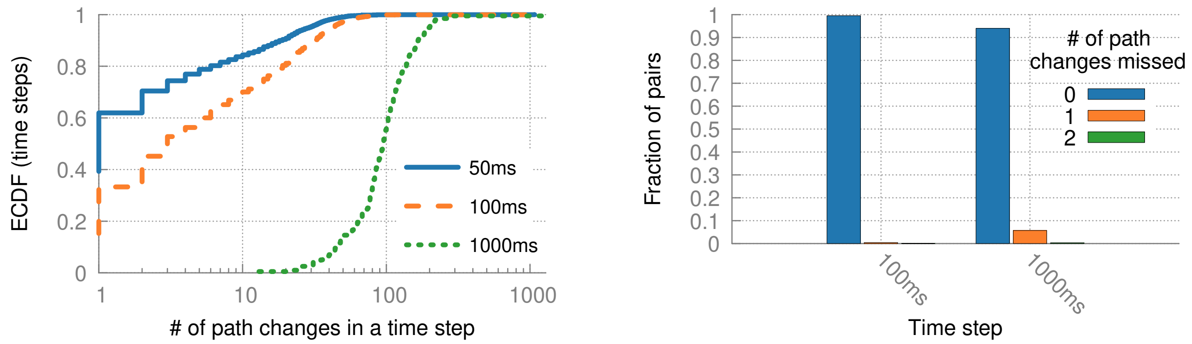


(a) *Distribution of path changes.* **(b)** *CDF across pairs: max RTT / min RTT.*

Figure 4.4: Endpoint path perspective. The paths between endpoint pairs experience significant amount of path changes and to lesser extent RTT difference over their lifetime.

experiences at least one path change (94% for Telesat, and 100% for Kuiper K1 and Starlink S1), with many seeing significantly more. This indicates that the LEO satellite network is highly dynamic, and that queue shifts are expected to happen not seldom but regularly. The same applies to the need to adapt to a varying RTT. Although RTT is relatively stable, which is expected given the path finding algorithm focus on latency minimization, still 5% (Telesat T1), 14% (Kuiper K1) and 30% (Starlink S1) experiencing an RTT 1.2× larger than their minimum RTT. This signifies that unlike static networks, there is significant variation in base RTT (*i.e.*, without congestion), thus highlighting the need for protocols aware and able to handle such lifetime differences. The differences between the constellations are due to their different operating height and their minimum angle of elevation. Further discussion regarding constellation differences can be found in the full publication [106].

The packet-level simulation makes use of dynamic state which is calculated on a certain time interval granularity. In order to investigate the required granularity, we perform the calculation at 50 ms, 100 ms and 1 s for Kuiper K1. For each timestep, we calculate the number of path changes (*i.e.*, from one time step to the other, how many paths had one or more hops different) (Fig. 4.5a) and as well the number of path changes missed compared to 50 ms granularity (Fig. 4.5b). The amount of path changes in each respective time step approximately follows the factor one would expect from 50 m to 100 ms, namely 2×. However, at 1 s granularity, a substantial portion of path changes were missed (6% missed one or more path changes), whereas at 100 ms only very few were missed (0.4%). Based on these results, we find that 100 ms is a good compromise



(a) *Approximately 2× more path changes in each 100 ms timestep and 10× in each 1000 ms timestep compared to 100 ms.*

(b) *1000 ms misses a substantial portion of path changes compared to 50 ms, whereas 100 ms does not.*

Figure 4.5: Time granularity for forwarding state updates.

between computational duration and simulation granularity.

4.4.4 Visualization

The final component of HYPATIA is its utility to visualize LEO satellite networks, and as such provide visual understanding to the network dynamics which occur. The visualizations were primarily the contribution of co-author Debopam Bhattacherjee. For completeness we provide a brief description of the types of offered visualizations. The full details and corresponding images can be found in the publication [106], as well as their interactive versions online [31]. There are four types of visualizations offered by HYPATIA, which are as follows. (1) Satellite trajectories visualize how the satellites move respective to one another in a constellation. (2) Views from the perspective of a ground station regarding the satellites that are visible to it and how long they are able to connect based on the minimum angle of elevation. (3) End-to-end paths which show the chosen (shortest) paths of communication between to end points. (4) Link utilization plots which show through color the utilization of (paths of) links.

4.5 Summary

To investigate the performance of networked systems, we require the right tools which model the most important characteristics. In this chapter, we have identified the use case of LEO satellite networks, for which existing tools are insufficient to capture its

intricacies to model congestion control, routing and traffic engineering. We presented the HYPATIA framework which addresses this need by enabling packet-level simulation over these dynamic LEO satellite networks. Its primary utility lies in modeling packet-level interaction, as well as its other two modules, general dynamics analysis and visualization, which are of significant value as they quantify the dynamic behavior and provide visual insight. We hope that HYPATIA can be used by network researchers to evaluate the applicability of existing methods, and to support the development of techniques able to tackle the many open challenges in LEO satellite networks.

5 RESOURCE ALLOCATION IN SERVERLESS QUERY PROCESSING

Serverless computing provides fine granularity time-based billing, low startup delay and rapid scaling. This is in stark contrast to more fixed infrastructure such as virtual machines, which often is billed with minima, takes longer to startup and thus scales slower. The works of Starling [158] and Lambada [147] show serverless to be an especially promising candidate to process interactive analytical queries (*i.e.,* finish in several to tens of seconds) on cold data (*i.e.,* infrequently accessed). The coldness of the data make it not worthwhile to be loaded into traditional databases, as that would lead to under-utilization and thus large expense. These serverless processing systems must be provisioned accordingly to achieve both good performance and cost-effectiveness. The extremes of allocation are not useful to the user: the cheapest option (small number of small functions) is too slow, and the fastest option (a large number of the largest functions) is too expensive. Rather, we would like an allocation which makes a compromise.

In this work, we set out to develop an advisory tool that can recommend suitable configurations for serverless query processing striking a good balance between completion time and financial cost. To do this, we propose a model similar to [125]'s approach for VMs, that considers the major factors affecting completion time and cost in serverless query data processing. It includes processing and network overheads (as [125], except using data and rate rather than CPU hours for processing), and as well accounts for startup time, exchange overhead, and request cost. The latter three factors are relevant for serverless data processing due to (a) the large number of workers involved, (b) the

brevity of interactive queries, and (c) communication through storage.

We use this model to augment an existing query data processing system, Lambada [147], with an advisory tool to provision serverless functions: given a query workload, the tool approximates the number of functions and their size to achieve a good balance between query completion time and the cost incurred. It does so by choosing configurations close to the "knee" of the Pareto frontier of these two dimensions. We show that obtaining accurate predictions of the run time or cost on serverless is very difficult due to a variety of system aspects that cannot be controlled by the user [88, 200]. However, we also show that it is still possible to recommend a reasonable configuration, thereby significantly simplifying the job of the user when using serverless for query processing over data lakes and expanding the narrow cost effectiveness of serverless query processing.

Chapter outline. In this chapter, we first present the niche use case of serverless data processing in §5.1. Second, we define the task of provisioning in §5.2. Third, we outline the challenges towards estimating serverless data processing in §5.3. Fourth, we describe the general estimation model in §5.4. Fifth, we describe the advisory tool and its evaluation methodology in §5.5. Sixth, we apply the model to the serverless query processing system Lambada [147] in §5.6. We then proceed onward with the evaluation on small benchmarks in §5.7 and a large benchmark in §5.8. Finally, we conclude with describing related work in §5.9 and briefly summarize the chapter in §5.10.

Contributions. Text and figures from the following publication were included in this chapter and its corresponding introduction and conclusion:

- Simon Kassing, Ingo Müller, and Gustavo Alonso. *Resource Allocation in Serverless Query Processing.* arXiv:2208.09519, 2022. [108]

5.1 Why serverless data processing?

Serverless has been proposed as an alternative to using virtual machines to process interactive queries on cold data [147, 158]. *Cold* data is so infrequently accessed that it is not cost-effective or practical to keep infrastructure on hot stand-by for processing it. Yet, users still expect *interactive* response times which requires sufficient computing power. For other query processing tasks, serverless might be too expensive. There are two reasons for this:

Billing method. Virtual machine instances are billed both at coarser time granularity and with a minimum time across all cloud providers. Both AWS [11] and Google Cloud [71] charge at a one second granularity, with a minimum of 60 seconds. Azure bills at either second [135], or minute [137] granularity—we could not find whether a minimum applies. Both factors substantially increase the cost of interactive queries whose completion time is in seconds. Suppose a query takes 6.5 s to complete. This would be billed 8% more expensive at one-second granularity, and 823 % more expensive at a one-minute granularity. In contrast, serverless functions are billed at very fine time granularity (AWS/Azure: 1 ms [14, 136], Google Cloud: 100 ms [70]) and with a small minimum billing duration per invocation (AWS: 0 ms [14], Google Cloud/Azure: 100 ms [70, 136]).

Startup delay. Due to data coldness, when an interactive query comes in, workers must be started as soon as possible in order to scale to process the data in a timely fashion. VM instances take significantly longer to start up in comparison to serverless functions, in the range of several tens of seconds to start up [82]. This is unacceptable for an interactive query that takes seconds to run. In contrast, serverless functions can start within a few hundred milliseconds [55] and a lot of effort is being invested in shortening that even further [5, 55, 198].

5.2 Provisioning and configuration

A user must decide what type of serverless workers to start and how many, in the same way as one would for a VM-based data processing system [77]. They differ however on the range of available configurations. For traditional virtual machine instances, cloud providers offer users the choice among a wide range of machine configurations, varying four key parameters [125]: number of cores (*compute*), network bandwidth (*network*), DRAM (*memory*), and SSD (*local disk size*). In contrast, serverless configurations are far more limited. Users can only select the memory size, from which some of the other properties are derived (Tab. 5.1).

	AWS	GCP	Azure
Memory	Configurable 0.125-10 GiB [14]	Configurable 0.125-8 GiB [70]	Configurable 1.5-14 GiB [136]
Compute	Based on memory (1769 MiB/vCPU) [12]	Based on memory [70]	Based on memory [136]
Network bandwidth	No information available [14,70,136]		
Local disk size (scratch)	Constant (512 MiB) [13]	Stored within the memory itself [68]	Constant (500 MiB) [140]

Table 5.1: ***Current serverless configuration offerings.***

Based on the current cloud offering, in this chapter we define the **provisioning of serverless data processing systems** as the choice of the **number of serverless workers** (W) and their **memory size in mebibyte** (M).

5.3 Estimation challenges

To recommend a suitable configuration, we build a model (§5.4) encompassing key characteristics of a serverless query execution system. The model does not try to make an accurate prediction of running time or cost but to differentiate between configurations. Estimating running time and cost in serverless is actually quite difficult (§5.7, §5.8) due to the many uncertainties associated with serverless execution [179,200]. In this section, we highlight both the fundamental prediction limitations and practical challenges inherent to serverless data processing to define the problem space.

5.3.1 Data-processing-related limitations

At the core of query optimization is the need to predict the cost of an operation, for which often metadata and statistics on the data (selectivity, distribution, etc.) are used. It is an open question how to do this over the vast amounts of data stored in a data lake. There have been considerable efforts, using zone maps [50] or gradual transformations of the

data [131], as well as table formats to include cost-based optimization metrics [21]. Lack of such data processing knowledge results in the first source of potential inaccuracies when trying to guess the running time of a query. In our case, these are the two factors that introduce the largest distortions:

- **Query selectivity.** The selectivity of an operator determines the amount of data the following operators process. As a result, the cost of intermediate operators in the query plan is difficult to gauge and can only be done on average or as a worst case.
- **Data distribution.** The performance of an operator (*i.e.,* process rate) is affected by the data distribution of the input. For instance, the population of internal data structures can result in longer processing time in aggregation operators relying on hash tables. If many input tuple keys are unique, the hash table will become full, causing collisions, resizing overhead, no longer fitting in caches, etc. This makes it difficult to estimate the cost of such operators even when they are on the leaves of the query plan. Moreover, the size of each individual tuple can vary; *e.g.,* 100 MiB of 10 byte tuples compared to 100 MiB of 100 byte tuples would result in 10$\times$ more operations.

We can account for these variances in two ways: (a) bounding the data size and effect on complexity, or (b) with additional knowledge about the data. In this project we opt for the former to avoid relying on functionality that is typically not available in data lakes. Note that adding such information (e.g., as suggested by [131]) to the model will only make it more accurate.

5.3.2 Serverless platforms related limitations

Serverless query processing systems make use of the many services a cloud provider offers, in particular, storage and serverless functions. The experienced quality of service, however, can vary: (a) the way the implementation and resource allocation of cloud services are opaque to their users, and (b) the resources behind these services typically are not reserved: they are shared among users and, thus, service quality can vary. The following are side effects observed in Amazon AWS influencing run time and cost:

- **Start-up time variance (*starting stragglers*).** The time from invocation until the worker is ready to start the computation is sometimes longer than usual, already creating *starting stragglers*. We hypothesize this can be an effect of the caching strategy, as well as temporarily contention due to high demand of the shared infrastructure.

- **Storage read/write variance (*storage stragglers*).** Serverless query processing systems make use of long-term storage (*e.g.*, S3 [16]) to read input and potentially write output, as well as to exchange data in-between computational stages. Infrequently, the read or write requests to S3 take considerably longer than expected (by as much as one order of magnitude). This can be due to a variety of reasons: *e.g.*, S3 replication, S3 caching, network packets being lost, or high demand (on either the function host or S3). Even if it occurs only 1% of the time, if there are 100 workers, these tail failure probabilities become significant [52, 158] and create large variations on the running time.
- **Queueing service send/receive variance.** Queuing services (*e.g.*, SQS) can be used to exchange small messages between workers, or to the driver, for example if the query result is small enough. Infrequently, but especially with many simultaneous finishes, the workers will simultaneously send their result message to the SQS queue. This can lead to significant delay if the underlying application/transport stack starts timing out. This is a known problem [9] in parallel processing when multiple parallel workers finish at the same time and all try to communicate back with a centralized server.
- **Computational contention.** A worker is allocated computational power proportional to its memory allocation. With limited computational power, in particular, less than one vCPU, even a single-threaded system combined with background computation can exhibit unpredictable computation patterns that can affect the observed network bandwidth and, thus, the time needed to read/write data.
- **Non-linear network performance.** In line with previous work [201], we observe that lambda functions in AWS can exhibit an increased network rate for a very short period of time at the beginning of their execution. Whether this behavior is seen or not depends on whether the storage (S3) can supply data at this increased rate and whether the function has sufficient parallelism available to deal with the higher incoming data rate (both in terms of vCPU and the need to download multiples files). In practice, it greatly affects the ability to estimate how long a query will take.
- **Polling requests.** When awaiting for a file from another worker to become available, a worker will poll to check when it is available. The more frequently the worker polls, the faster it is made aware and can start reading; however, this affects the price since, the more read requests, the higher the cost of the computation. The duration of waiting (and thus polling) is influenced by stragglers.

Because of these intrinsic difficulties, in this project we do not aim to build a query cost

predictor. Our model does accurately estimate running time and cost in many situations but not always. However, the estimations are good enough to be able to differentiate between configurations in terms of picking suitable ones in terms of balancing running time and cost. This is the basis for the advisory tool we propose.

5.4 General estimation model

Rough estimations of cost/execution time on virtual machines have been recently proposed [125]. However, such an approach does not directly translate to serverless functions due to: (a) the difference in the number of workers involved (10s for virtual machines, 100s for serverless functions), (b) the overhead incurred for data exchanges (typically through storage), and (c) the difference in task duration. Serverless operates in the few to tens of seconds range, whereas virtual machines in minutes to hours – factors insignificant at a one hour scale become so at a one second scale [179, 200]. In this section, we formulate an analytical model which incorporates the most important factors in our understanding that effect the completion time and financial cost of query processing using serverless functions. The model includes processing and network overhead similar to [125] (using data and rate instead of CPU hours for processing), but also incorporates the factors unique to serverless query processing systems, namely startup time, exchange overhead and request cost. The model we present is directly applicable to Lambada [147], and we posit models in a similar vein can be applied to other systems to roughly estimate performance albeit with changes, for instance to account for different number of workers per stage (*e.g.,* for [112, 158]).

5.4.1 Start-up

The largest serverless function is equivalent to at most a medium-tiered virtual machine instance (10 GiB of memory with 5.8 vCPUs for AWS). Thus, many serverless functions must be spawned to be able to process large amounts of data. A single driver to start the workers can only start workers at a limited rate. To speed up the start up phase, it has been proposed to use a two-level broadcast tree where the first workers start other workers [147]. We model the starting of serverless workers by an invocation rate $R_{\text{1-inv}}$ and $R_{\text{2-inv}}$ (*i.e.,* how many workers can be started per second) for the main driver and a worker respectively. We define the invocation delay as $T_{\text{inv. delay}}$ (*i.e.,* how long from

invocation till the worker is started). For each worker $i \in 1..W$, we estimate its startup and ready time.

One-level invocation:

$$T_{\textit{startup}}(i) = \frac{i}{R_{\textit{1-inv}}} + T_{\textit{inv. delay}}$$
$$T_{\textit{ready}}(i) = T_{\textit{startup}}(i)$$

Two-level invocation:

$$T_{\textit{startup}}(i) = \begin{cases} \frac{i}{R_{\textit{1-inv}}} + T_{\textit{inv. delay}} & i \leq \sqrt{W} \\ \frac{g}{R_{\textit{1-inv}}} + T_{\textit{inv. delay}} + \frac{h}{R_{\textit{2-inv}}} + T_{\textit{inv. delay}} & \textit{else} \end{cases}$$

$$T_{\textit{ready}}(i) = \begin{cases} T_{\textit{startup}}(i) + \frac{\sqrt{W}}{R_{\textit{2-inv}}} & i \leq \sqrt{W} \\ T_{\textit{startup}}(i) & \textit{else} \end{cases}$$

g is the index of the worker that starts i and h the index within worker g's start list (first $\sqrt{W}$ workers invoke $\frac{W-\sqrt{W}}{\sqrt{W}}$ others each). The *startup* time indicates when the worker has started, and *ready* time when the worker can start its data processing tasks (both since query epoch). This is different for some workers in the two-level invocation, as they must first invoke other workers—which is part of the billed time.

5.4.2 Base overhead

Every worker performs a certain amount of base tasks that are independent of the workload, including initializing its variables, extracting input, allocating memory, and packaging its output. We model this (billed) overhead time as a single constant $T_{\textit{base}}$.

5.4.3 Process, compress, and network rates

We model the process, compress, and network rate as three constants (dependent on the memory size chosen): $R_{\textit{network}}$, $R_{\textit{compress}}$, and $R_{\textit{process}}$. We consider network and compress rate as independent of the workload, whereas the process rate as dependent. These constants can be attained through measurement (as is done in §5.6.3), by applying some model (*e.g.,* using Amdahl's law) to account for the effect of more vCPU, or a

combination of the two. Network rate is the rate at which both transfer of read and written data takes place.

5.4.4 Input

The input is solely concerned with the reading and processing of the input:

$$T_{input} = max(\frac{D_{input}}{R_{network}}, \frac{D_{input}}{R_{process}})$$

5.4.5 Exchange(s)

The purpose of an exchange operator is to partition data such that each function receives all tuples with the same key. An exchange can be done in one or more levels, which for two or more levels means that first functions exchange in a group, and then having the groups exchange (and so forth depending on number of levels). An exchange operator is used to accomplish join and reduce operations, among others. Its operation consists of compressing and writing the data, and subsequently reading and processing the data destined to the function. The number of the group size in an exchange level is defined as $G_{ex[j]}$, a value we calculate based on the total number of workers and whether an exchange is one or two levels. We define an additional penalty factor for each worker participating in the exchange level, $T_{overhead[j]}$. For each exchange level j:

$$T_{ex[j]} = \frac{D_{ex[j]}}{R_{compress}} + \frac{D_{ex[j]}}{R_{network}} + max(\frac{D_{ex[j]}}{R_{network}}, \frac{D_{ex[j]}}{R_{process}}) + G_{ex[j]} \times T_{overhead[j]}$$

5.4.6 Output

The output is solely concerned with the amount of data to compress and subsequently written to storage:

$$T_{output} = \frac{D_{output}}{R_{compress}} + \frac{D_{output}}{R_{network}}$$

5.4.7 Postprocessing

The driver must receive the final results from the workers and process them to return a final result to the user. We model this as a constant: $T_{postprocess}$.

5.4.8 Requests costs

Unlike virtual machines, which almost exclusively communicate directly between each other, most serverless data processing systems communicate through storage services (*e.g.,* S3 [147, 158]). In general, cloud providers charge for the access to the storage service. In exchanges between many participants, the number of requests can become a significant cost factor. When applying the advisory tool to a system, we assume we can give a reasonable estimate of the request cost $C_{requests}$.

5.4.9 Overall Completion Time and Cost

The completion time is defined as the last worker completing and its result having been collected by the driver:

$$T_{completion} = T_{ready}(W) + T_{base} + T_{input} \; + \\ (\sum_j T_{ex[j]}) + T_{output} + T_{postprocess}$$

The billable time of each worker does not directly equal the completion time. Instead, it equals the sum of time the worker was alive:

$$T_{billable} = \sum_i T_{alive}(i)$$

The alive (*i.e.,* billable) time of a worker is only independent from other workers if there no exchanges as else it has to pace with the slowest worker. We define the worker alive time of a simple scan (without exchanges) as:

$$T_{alive}(i) = T_{ready}(i) - T_{startup}(i) + T_{base} + T_{input} + T_{output}$$

Similarly, if there are exchanges, the workers have to wait until the last worker is ready. Hence, we define the worker alive time for a query with exchanges as:

$$T_{alive}(i) = T_{ready}(W) - T_{startup}(i) + T_{base} + T_{input} + \sum_{j} T_{ex[j]} + T_{output}$$

The final cost is the sum of the worker runtime cost and the sum of the requests cost:

$$C_{total} = T_{billable} \times price/ms + C_{requests}$$

5.4.10 Model analysis

The general model has several parameters, each of which effects increase in completion time and/or financial cost, the latter either through increased billed serverless time or requests cost. The general tendencies are as follows. Firstly, increasing the number of workers (W) can have the following effects:

- increased completion time as more workers must be invoked and exchange overhead is larger;
- decreased completion time as the amount of data each worker reads, writes and exchange reduces;
- increased cost through billed time as there is base overhead and invocation;
- and increased cost through request cost as there are more exchange members.

The above effects are under the assumption of invocation and exchange levels remaining the same. Increasing the worker memory size (M) can have the following effects:

- decreased completion time as each worker can network send/receive, process and compress at a higher rate;
- increased cost through billed time as each time unit is more expensive.

As these examples show, several effects on completion time and financial cost compete against each other when increasing the number of workers (W) and memory size (M), increasing the intrinsic difficulty of estimating either.

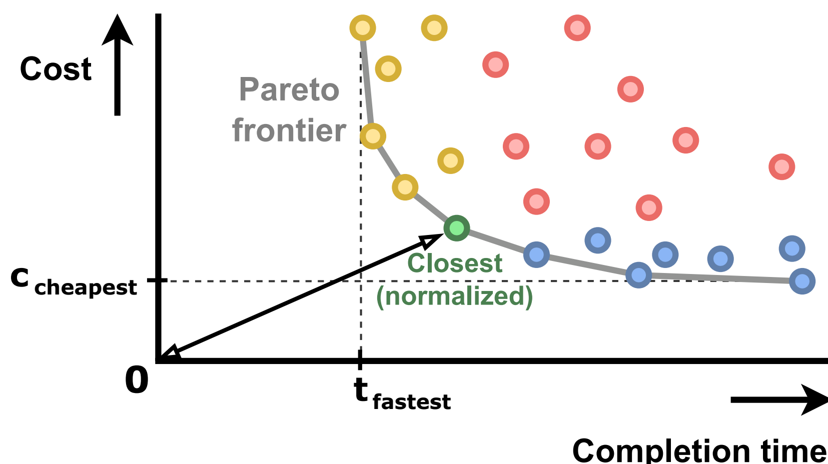


Figure 5.1: Pareto plot of completion time and cost. The circles are candidate configurations (the coloring corresponds to Fig. 5.2). The *knee-opt* is the configuration closest (normalized) to the optimum of cheapest cost and fastest completion time.

5.5 Advisory tool and its evaluation methodology

The advisory tool we propose uses the estimation model just presented. The tool aims to select a configuration balancing completion time and cost as there is no configuration (*i.e.*, number of workers W and their memory size M) that minimizes both. In the next sections we describe how we define the best configuration (§5.5.1) and how we evaluate the advice (§5.5.2).

5.5.1 The Pareto knee

The choice of configuration is a multi-objective optimization of completion time and cost, and as such can be visualized as a two-dimensional Pareto plot (Fig. 5.1). We consider as suitable configurations those closest to the "knee" of the Pareto frontier. We automate the picking of this "knee" using the following method.

Let F be the set of candidate configurations. For each candidate configuration $i \in F$, we define its completion time as t_i and its cost as c_i. We select as optimum the configuration with the shortest distance to the origin, with its completion time and cost normalized by the fastest and cheapest possible outcomes across all configurations:

$$t_{\textit{fastest}} = min_{i \in F}\ t_i \qquad c_{\textit{cheapest}} = min_{i \in F}\ c_i$$

$$d(i) = \sqrt{\alpha(\frac{t_i}{t_{\textit{fastest}}})^2 + \beta(\frac{c_i}{c_{\textit{cheapest}}})^2}$$

$$\textit{knee-opt} = arg\, min_{i \in F}\ d_i$$

The selection of this point is depicted visually in Fig. 5.1. In our work, we set the weights α and β to one, which means that we value completion time and cost equally. For example, a configuration with $2 \times t_{\textit{fastest}}$ and $1 \times c_{\textit{cheapest}}$ has the same score as a configuration with $1.6 \times t_{\textit{fastest}}$ and $1.6 \times c_{\textit{cheapest}}$ approximately.[1] A related work on multi-objective optimization for cloud data analytics proposed a similar method, but instead uses as metric the Euclidian distance directly to the idealized fastest and cheapest point (the so-called "Utopia point") [185] (further discussed in §5.9). We chose to use the normalized distance to the origin such that the resulting distance metric is relatively interpretable and comparable.

5.5.2 Advisory tool and evaluation

The advisory tool uses the procedure above to select a configuration. As it does not know the actual completion times and costs, it instead uses the values supplied by the estimation model.

In the later experiments, we run all configurations in the Cartesian product of two reasonable candidate lists of number of workers and memory size based on the partitioning and the query's operators' in-memory requirements, respectively. This provides us the data to determine the experimentally best choice of configuration. We use this as the baseline to compare the advice against. The primary comparison metric is the relative difference between their distance metric. Two additional metrics we use are the signed relative actual completion time and actual cost error of the advised configuration to the optimal configuration. If the advisor chooses the actual optimal configuration, both errors are zero. A negative relative error means that in that one dimension, the configuration selected was better. At most one of the two dimensions (time or cost) can have negative relative error as the optimal configuration is part of the Pareto frontier.

[1] As $\sqrt{5/2} \approx 1.6$

Relative actual completion time error (columns) / **Relative actual cost error** (rows)

	Positive	Zero	Negative
Positive	Slower and more expensive	More expensive with no speed-up	Faster, but at too high price
Zero	Slower with no reduction in cost	Correct advice	
Negative	Cheaper, but at too much slowdown		

Figure 5.2: The advisory tool's possible outcomes. Because we compare actual outcomes, it is not possible to outperform the optimal configuration simultaneously in both dimensions as it is part of the Pareto frontier.

The possible outcomes of an advice issued by the configuration advisor are shown in Fig. 5.2.

To illustrate, consider the following example. Suppose the best configuration has $t = 1.1 \times t_{fastest}$ and $c = 1.3 \times c_{cheapest}$, whereas the advised configuration outcome has $t = 1.4 \times t_{fastest}$ and $c = 1.2 \times c_{cheapest}$. The distances of the best and the advised configurations 1.70 and 1.84, respectively. The aforementioned metrics are in this case the following: the advised configuration is overall $\frac{1.84-1.70}{1.70} = 8\%$ worse than the experimentally determined best choice and it runs $\frac{1.4-1.1}{1.1} \times 100\% = 27\%$ slower at an $\frac{1.2-1.3}{1.3} \times 100\% = -8\%$ cheaper cost.

5.6 Applying the model: Lambada

To test the tool, we use it on Lambada [147]. We first describe Lambada (§5.6.1), and then explain the experiments conducted (§5.6.2). We apply the general model by filling in the Lambada/AWS-specific parameters (§5.6.3) and finally analyze the effect of the filled values in terms of overarching configuration preference (§5.6.4).

5.6.1 Description

Lambada [147] takes as user input a computational DAG of operators, from which it devises a distributed query execution plan. The driver starts the selected amount and type of serverless workers (*i.e.,* AWS Lambda) and passes on the (distributed) plan to each worker. If many workers have to be started (if $W > 100$), the driver makes use of "two-level invocation": workers started first are also in charge of starting other workers. Once a worker is ready, it starts executing the plan: each worker reads its data partitions from cloud storage (*i.e.,* AWS S3). The partitioned tables are stored in Parquet format, to take advantage of compression and the ability to only download the columns relevant to the query. Each worker then goes through the same computations (albeit on different data), and amongst them perform exchanges when data should be shared (*e.g.,* join, aggregation). Exchanges take place through S3: each worker writes to a file the data to send to other workers. The exchanges can either be one-level ("all-to-all") or two-level (if number of workers $W > 32$). Two-level exchanges double the amount to read and write, while quadratically reducing the number of read and write requests (which is what the cloud provider charges for). Once done, a workers returns its result by putting them in a shared result queue which the driver monitors. The driver collects all results, performs some last operations, and returns the result to the user.

5.6.2 Instrumentation

To capture runtime and cost information, we have augmented Lambada with the following instrumentation:

- **Query completion time.** The completion time is measured by the driver from the moment it starts the workers until it has returned the final result to the user.
- **Query cost.** Each worker keeps track of the time from when it started until it returns the final result to the SQS queue, which constitutes the amount of billable time. Each worker counts the number of HEAD, GET, and PUT requests and logs them.

We use worker logs to keep track of request counts but also for post-mortem analysis: to know how long certain operations and functions within the workers take. This is vital to identify bottlenecks and the occurrence of stragglers. The logs are not used by the system itself, and as such the upload (by the workers) and download (by the driver)

of worker logs are excluded from both the billable time of the workers and the final completion time.

For network stability reasons, we use an AWS instance as the driver rather than a machine in our local cluster. We chose a `m5a.2xlarge` instance as it is comparable to a high-end laptop with its 8 vCPU and 32 GiB of memory. We deploy our instance in the `eu-west-1` region, where we also store the S3 data used for the experiments and launch the workers.

In the running of (micro)benchmarks (§5.6.3, §5.7, §5.8), Lambada exhibited in a very small set of the runs internal errors that caused the run to fail (in a probabilistic fashion), among which: the lack of idempotency when handling SQS messages (*i.e.,* failing in the extremely rare case a message is received twice if a delete fails), SSL validation errors due to too many concurrent invocations (*i.e.,* the SSL request times out), and in some rarer cases there were certain (likely network-related) exceptions that were not caught. In the few cases in which any of these occurred, they were rerun.

There is another class of failure, which is to be expected: it is possible for Lambada to fail because a worker ran out-of-memory or worker start-up reaching a time-out limit of 20s. We consider a run for which this happens (one or more times among its repeats; generally it is for all instances) an infeasible configuration.

5.6.3 Applied model

In this section, we fill in the parameters of the general model (§5.4) except the data sizes, which we fill in on a per-query basis in the experiments (§5.7, §5.8).

Invocation. The rate and delay at which a system can invoke workers depends on the invocation implementation and the cloud. We use 128 concurrent threads for invocation on the driver, and 32 on a worker. We perform a microbenchmark in which we start 512 workers using only the driver or only the worker, under either COLD start[2] and HOT start.[3] Tab. 5.2 shows the mean invocation rates and the mean 99th percentile invocation delay across three repetitions. The later experiments of this project all are HOT started, in order to have the many runs finish in feasible time (as re-deployment to achieve coldness takes long and is rate-limited by AWS).

[2] By re-deploying the function beforehand to ensure it is not cached.

[3] By performing another run with at least as many workers beforehand, which warms up the AWS lambda caches.

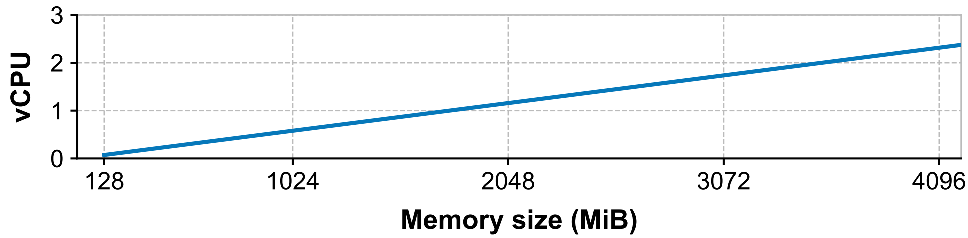


Figure 5.3: vCPU allocation on AWS (linear scaling with 1769 MiB/vCPU up to 10240 MiB).

Base overhead. We perform a simple benchmark in which each worker retrieves a single small file. We subtract the scan duration from the total time the worker is deducted from its alive time. The result, its mean across three repetitions, is made monotonic, and is shown together with the measurements in Fig. 5.5.

Compress rate. Before writing data to storage, it must first be compressed into Parquet format. We assume the compression rate is independent of the type of underlying data. We perform an exchange microbenchmark with each of 32 workers reading and compressing 32 MiB with two columns. The experiment outcome is shown in Fig. 5.4(top). The mean compression rate (for all 96 samples across three repetitions) made monotonic is used in the model, depicted in Fig. 5.4(bot). It scales proportionally with vCPU allocation (Fig. 5.3) (single-core bound).

Network rate. Unlike VMs [125], there is no network bandwidth info available for serverless functions (see Tab. 5.1). Although it is observed that there is an initial higher network burst rate [147, 201], we model network rate at its sustained state. In the microbenchmark, each worker reads in a 128 MiB file with eight columns. The mean network (for all 96 samples across three repetitions) made monotonic is used in the model, depicted in Fig. 5.4(bot). Note that especially network rates can have outliers, we have observed very low rates every now and then (see minima in Fig. 5.4(top)). Lambada has a small multi-threaded behavior in which, if it has two partitions to read in, it does so simultaneously. If it has sufficient compute available to it (at least more than 768 MiB), it can achieve (temporarily) a network rate higher than the model predicts. We consider these conditions too system-specific and as such are not captured in the general model (in this microbenchmark, there is only a single input partition per worker).

Requests and exchange overhead. Lambada fetches the exact columns it needs from

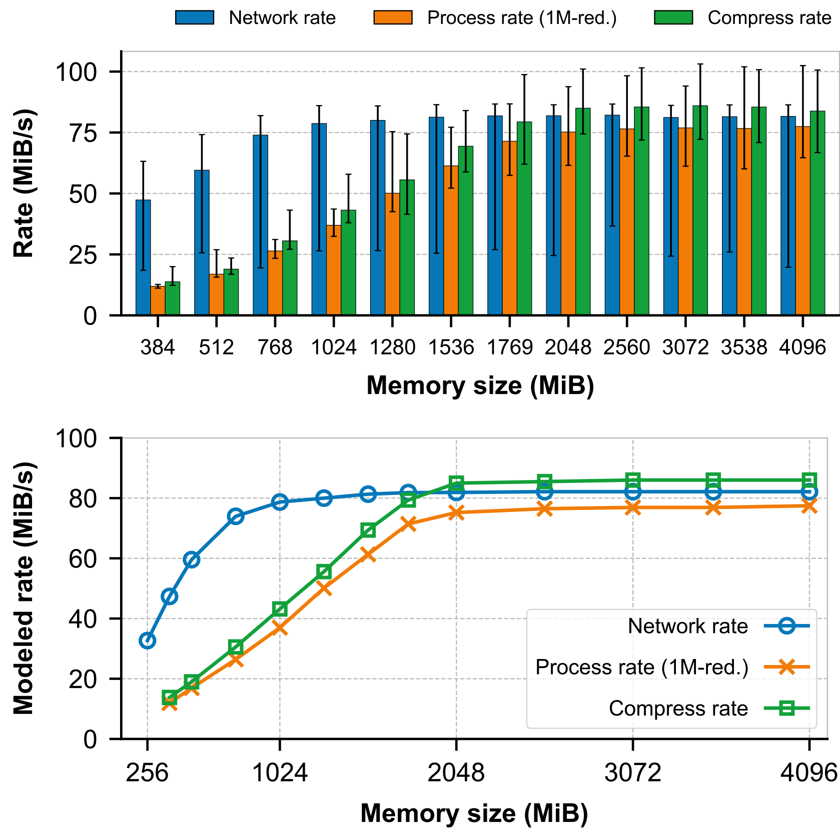


Figure 5.4: Network, process and compress rate (top: measurements, bottom: monotonic model). Error bars in top bar plot are min-max.

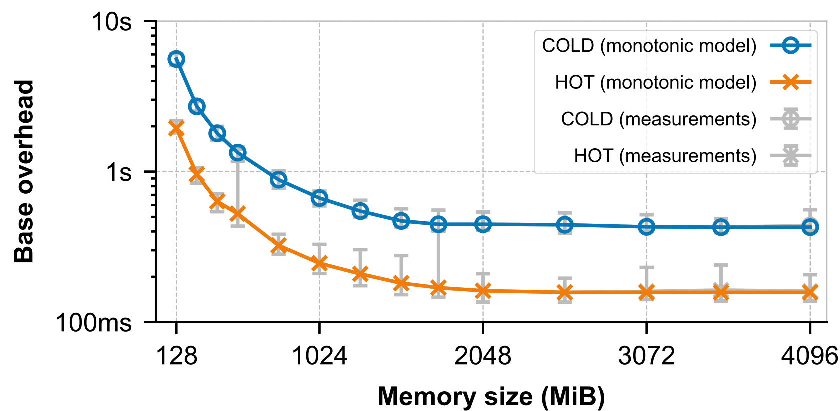


Figure 5.5: Base overhead. Error bars are min-max.

Parameter	Value
Driver invocation rate $R_{1\text{-}inv}$	142.3 inv/s
Worker invocation rate $R_{2\text{-}inv}$	93.6 inv/s
Invocation delay $T_{inv.\ delay}$	COLD: 1263.4 ms, HOT: 677.7 ms
Postprocess duration $T_{postprocess}$	188.5 ms
Overhead per exchange member $T_{overhead[j]}$	Scales linearly with the number of GET and HEAD requests

Table 5.2: Memory-independent cloud-/system-parameters.

each Parquet file. With the knowledge of tables/columns involved for each input scan and exchange, we can calculate the exact number of GET requests. Due to availability polling during exchanges, the number of HEAD requests can vary between runs. An exchange microbenchmark yielded approximately 3 HEAD requests for each exchange file. We use the number of requests and their duration to determine the exchange overhead. Microbenchmark measurements indicate a HEAD and a GET request take approximately 13.1 ms and 18.4 ms on average, though there were values of up to 215 ms (HEAD) and 530 ms (GET). The exchange overhead per group member $T_{overhead[j]}$ is set to the multiplication of the requests to read its exchange files and the request type duration. From the AWS documentation, we directly retrieve pricing for serverless function time (0.0000166667 $ per GiB-second) [14], and read and write requests (respectively 0.0004 $ and 0.005 $ per 1000 requests) [17].

Post-processing. We have 128 workers each fetch a file of 32 MiB, and measure the duration between the last worker having returned its result and the driver finishing. Tab. 5.2 shows the mean duration across 10 repetitions.

Processing rate. The process rate of a query is the most difficult parameter to estimate, as it is query dependent. As an admittedly imperfect solution, we use the reduction to 1 million unique values as the process operation; its internal data structure, the hash table, is used across many common operations such as joins, reductions, and group-by's. The results are shown in Fig. 5.4(top). Note that Lambada's operator execution model is single-threaded and only marginally benefits from multiple vCPUs: it roughly scales proportionally with vCPU allocation (Fig. 5.3) up until single core.

5.6.4 Applied model analysis

The general model applied to Lambada (§5.6.3) preserves the general model influences (§5.4.10) by monotonically modeling the rates and overhead. Its modeled sustained network rate is capped out at 82.1 MiB/s, with 78.7 MiB/s being already reached at $M = 1024$ (0.58 vCPU). In addition, due to its mostly single-threaded nature, its process rate and compress similarly are capped at respectively 77.5 MiB/s and 86.0 MiB/s, with 75.2 MiB/s and 85.0 MiB/s being already reached at $M = 2048$ (1.16 vCPU). This means that:

- if network is the bottleneck, increasing the memory size beyond $M = 1024$ has little effect on completion time;
- if compute is the bottleneck, increasing the memory size beyond $M = 2048$ has little effect on completion time.

5.7 Into practice

In this section, we describe the manner how to apply the model and the practical guidelines in terms of choosing the correct configuration that it implies. We first describe in §5.7.1 how we apply the evaluation methodology of the advisor (§5.5) in our benchmarks. In the context of serverless data processing tasks, we characterize workloads into two categories: (1) workloads that solely scan data (§5.7.2), and (2) workloads that involve one or more exchanges (§5.7.3). Either category has its own trend in the sweet spot in its choice of number of workers and memory size. This section lays the groundwork for the large scale benchmark in §5.8.

5.7.1 Methodology

The goal of our work is to characterize serverless data processing performance in general, which implies some version of ideal circumstances. As such, we want to evaluate the model's rough estimation and advice using an ideal system. In particular, this means we do not wish to compare to runs that are unfortunate enough to encounter stragglers (§5.3), which are in particular present for network rates and requests (§5.6.3, §5.6.4). We

similarly can only perform a limited number of repeated runs with a Cartesian product of memory sizes and number of workers. In order to eliminate outliers, we opt to choose the run best suited to be a kneepoint. We choose this by first performing the distance metric using all data points, and for each configuration choose the run which had the smallest distance metric. The chosen configuration is then the experimental best and used as a reference for error calculation. As such, each point in any plotted actual Pareto frontier (*e.g.,* Fig. 5.6b) is not the mean across all repetitions, but the best run among them. Note that although we do choose the representative data point for each configuration to be the best, the normalization is done with the cheapest and fastest outcome across all repetitions. This is done because the representative data point selection is to account for outliers, whereas normalization should be done with the true lowest values. The theory as presented in §5.5.2 is upheld, as the fastest and cheapest value can only be lower by considering all repetitions.

It is possible that the advisor recommends a configuration which does not have an actual experimental outcome (*i.e.,* due to running out of memory or hitting the timeout limit), as it is not always aware of this. In this case, which is exceedingly unlikely occur, the advisor error would be set to infinite – though, in none of our experiments this was the case as these are generally the extremes.

5.7.2 Scan workloads

In a **scan workload**, there is no communication between workers. Workers read in the input partitions assigned to them, scan through their assigned data, and return their result directly to storage or the driver in case of simple aggregation. The typical use case for this workload type is to select tuples that fit certain criteria, or very simple aggregations whose return value is a few tuples (*e.g.,* counting, summing, reduction to very few tuples). The associated computational requirement is as such not much. The primary bottleneck is their reading of input over the network.

Based on the applied model analysis (§5.6.4), around a memory size of 768-1024 MiB there starts to be diminishing returns on network rate when increasing further. Because the workers do not exchange data in a scan, there is no strong penalty on the increase of workers to large numbers. Only the relatively minor base overhead and invocation increase the billed runtime, and only the latter to a small extent the completion time (as more workers must be started).

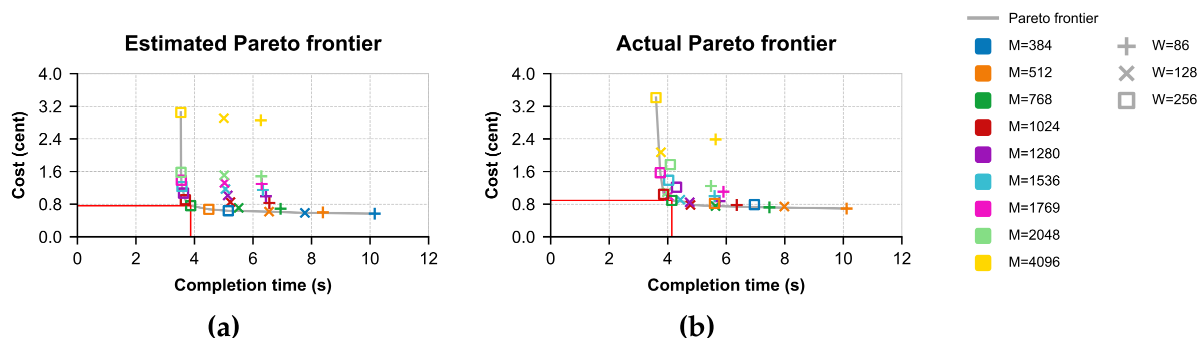


Figure 5.6: Scan benchmark: Pareto frontiers with the red lines pinpointing their kneepoint (estimated = actual: W=256, M=768).

To study the scan workload, we perform the following experiment. We make use of a 32 GiB table of eight 64-bit columns filled with random values. It is partitioned into 256 Parquet files of 128 MiB each. The query counts the number of rows (which is 2^{29}). As this operation is computationally simple, we set the model $R_{process}$ to a value larger than $R_{network}$, which means it is of no consequence. As only the large number of workers are of interest, we vary the number of workers $W \in \{86, 128, 256\}$ (corresponding to 3, 2, and 1 partitions/worker respectively). We vary the memory size at steps between 384 MiB and 4096 MiB. The configurations (W=86/128, M=384) did not make the 20 second start-up time limit and as such did not finish. Decreasing the number of workers further results in too long completion time, and decreasing the memory size further leads to out-of-memory (*i.e.,* infeasible) and too slow completion. Both of which are not part of the regime of interest, which is the knee of the curve: achieving decent performance at a cost-efficient price point. Each configuration run is repeated five times.

The estimated Pareto frontier is depicted in Fig. 5.6a: the maximum number of workers $W = 256$ with a modest memory size of $M = 768$ is advised to achieve both good completion time and cost. This advice matches the experimentally determined best choice, as is seen in the actual Pareto frontier in Fig. 5.6b.

Note that the estimation does not always match the actual outcome. For $W = 256$ the completion time and cost estimate is slightly lower: this is due to regular variance of network rate. For $W = 128$ and $W = 86$ the estimates are consistently too high in completion time and cost for memory sizes over 768 MiB: this is because $W = 128$ and $W = 86$ have more than one input partition, such that Lambada can download two input partitions simultaneously. With its dual download and sufficient compute to it, it can make use of the temporary burst rate serverless functions on AWS have available to

it [147]. Our model asserts only a sustained rate, which purposefully does not capture such system-specific intricacies.

These results show that scan workloads are generally network-intensive, not compute-intensive (unless the scan involves a computationally heavy operation *e.g.*, a reduction-by-key), and do not include exchanges. Their optimal configuration is to make use of many workers with a memory size that provides high network bandwidth.

5.7.3 Exchange workloads

In an **exchange workload**, workers read in their respective input data, and then proceed to perform one or more exchanges to facilitate certain global operations such as joining, grouping, or sorting. The associated computation can be considerably intensive, and its complexity can depend on the data (§5.3.1). An exchange workload has to not only read input data and write output data, but also read and write for each exchange. As such, it is network-intensive from a rate perspective. Besides rate limits, there is also a cost to an increase in exchange members: there is a limit to the amount of outstanding requests and ongoing connections. This is particularly important for Lambada, which communicates through the writing and reading of files to storage (a practice not unique in serverless data processing systems [158]) as inter-lambda communication is not (easily [201]) possible. For every exchange member, a worker has to read in all columns individually.

Following the applied model (§5.6.3), we expect the memory size with near-highest network rate, process rate, and compress rate to be the best pick: around M=1769-2048 appears to be the best. We expect a number of workers that balances for completion time the decreasing influence of more workers processing the amount of data (*i.e.,* rate) and the increasing influence of the extra overhead caused by more exchange members. Another factor we must consider is out-of-memory: hash tables can grow to large sizes with many unique keys for instance.

We examine the exchange workload through a benchmark. We use a 4 GiB table of one column filled with 64-bit random values, partitioned into 256 Parquet files of 16 MiB each. The query is to count the number of unique rows (which is in expectation 2^{29}) – which requires a reduction operation. This is a rather compute-intensive operation: every tuple processed will be inserted into the hash table that facilitates the reduction. We take the 1M-reduction ($\approx 2^{20}$) processing rate from Fig. 5.4, although the operation will likely be slower. This is one of the fundamental limitations of estimating arbitrary compute: it is

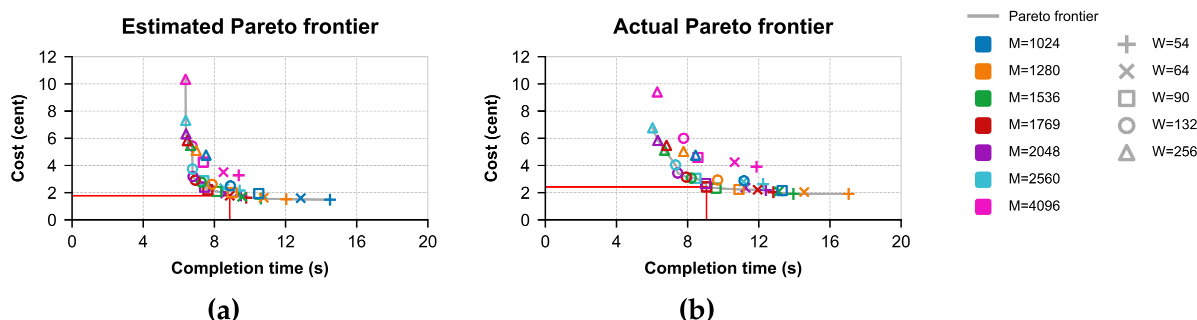


Figure 5.7: Exchange benchmark: Pareto frontiers with the red lines pinpointing their kneepoint (estimated: W=64, M=1769, actual: W=90, M=1769).

as arbitrarily difficult. We vary the memory size between 1024 and 4096 MiB in several steps. We choose a number of workers in the mid-range, with 1 to 5 partitions/worker: $W \in \{54, 64, 90, 132, 256\}$. Reducing memory size or number of workers further results in out-of-memory cases in the Cartesian product. The configurations (W=54/64, M=1024) already ran out-of-memory and as such did not finish. Some of the number of workers (54, 90, 132) are slightly higher than the partitions/worker requires: this is to have a performant two-level exchange as we must divide the workers in two groups X, Y such that $X \times Y = W$ exactly while minimizing $X + Y$ to get good performance.

The estimated Pareto frontier (Fig. 5.7a) generally underestimates the completion time and cost when compared to the actual Pareto frontier (Fig. 5.7b). This is caused by the unexpectedly slower processing experienced in practice. Nevertheless, what is important for the advisor is that the shape, the relative positioning of configurations to each other in completion time and cost, is preserved. The advisor recommends the (W = 64, M = 1769) configuration as best, whereas the experimentally determined best choice actually was the (W = 90, M = 1769) configuration. If the user would have followed the advice (as in practice, the user would not explore the space), there would have been a 32% increase in completion time with a -9% cost reduction. With additional prior information (*e.g.*, if the query is run repeatedly), this discrepancy could be adjusted and the estimation (and as such, the advice) improved. For example, by simply reducing the model process rate by one-third for all memory sizes in Fig. 5.4, the advisor would elongate along the x-axis (completion time) and correctly pick the best configuration.

Unlike for scanning, an increase in workers leads also to an increase in the number of requests ($\mathcal{O}(W\sqrt{W})$ [147]) and exchange overhead. These two factors cause the curve to go up with the number of workers: for instance, at (W = 64, M = 1769) a mere 15%

SF	Partitions	Total size (GiB)	Number of workers range	LINEITEM (MiB)	ORDERS (MiB)	PARTSUPP (MiB)	CUSTOMER (MiB)	PART (MiB)	SUPPLIER (MiB)
20	39	6.5	2, 3, 5, 10, 13, 20, 39	109.6	29.2	21.7	6.3	3.4	0.4
100	192	33.0	6, 12, 24, 48, 64, 96, 192	113.4	30.4	22.1	6.4	3.4	0.4
200	384	67.0	12, 24, 48, 96, 128, 192, 384	114.2	32.1	22.1	6.4	3.4	0.4
500	960	168.7	30, 60, 120, 240, 320, 480	115.0	32.6	22.1	6.4	3.4	0.4

Table 5.3: ***Large benchmark datasets of different scale factor (SF) used in the experiments: for each table, the measured mean Parquet partition file size is stated. REGION and NATION are omitted as their (small constant) size does not increase with SF.***

of cost is requests, whereas at (W = 256, M = 1769) it is 38%. Note that the benchmark exchange has a large amount of exchanged data (its entire input). In other exchanges which exchange little data, throughput is likely to be less of a bottleneck, whereas the exchange overhead can dominate the completion time.

In exchange workloads, the optimal configuration makes use of a number of workers that balances total rate and the overhead/cost of the enlarged exchange. Exchange workloads are generally both network- and compute-intensive, and as such favors a memory size which offers both at reasonable cost.

5.8 Large benchmark

A version of the TPC-H benchmark [192] that runs on Lambada is used to explore how well the estimation performs for representative workloads [147]. Here we expect to see larger errors in estimating running time and cost but what matters is how close the suggested configuration is to the optimal.

5.8.1 Dataset generation

For each scale factor (SF), we performed the following steps to prepare the dataset for consumption by lambda workers. (a) We generated the raw text CSV files using the Lambada [147] version of dbgen [193]. (b) We use LINEITEM as the base to decide how to split all tables as it is the largest table. At SF=1, the size of LINEITEM is approximately 586.2 MiB uncompressed. After compression into a Parquet file, it is 193.3 MiB. We set the divisor to achieve a size of $\pm$ 100 MiB per LINEITEM partition. We split each single large CSV file (of the eight tables) into the smaller files. (c) We convert each raw text CSV to Parquet format using Apache Arrow [20]. The final result for a subset of the scale factors is depicted in Tab. 5.3. (d) The files are stored on AWS S3. Note that in query execution, only the relevant columns are read: a worker retrieving a partition thus only retrieves a fraction of the total size of each file. We did not sort the CSV files before splitting, so the min/max column metadata in the Parquet files does not allow reads to be skipped.

5.8.2 Query modelling

For each query, the model workload parameters must be determined to facilitate the rough estimation. The **selectivity** in the data flow we manually determine by loading the benchmark data (at SF=1) into a relational database, and manually executing selection queries corresponding to each phase (input, exchange(s), output). For example for Q4, approximately 63.2% of the LINEITEM tuples and 3.8% of ORDERS tuples pass the input selection criteria. Q19 of TPC-H was altered using the correct REG AIR instead of AIR REG. The compression ability of Parquet varies for each column, and the workers only read in the columns relevant to the query. To determine the **tuple data size** of a read, we determine the mean entry size of each column by inspecting the Parquet files. For each table, we inspect the first partition at SF=100, which yields us the total byte size of each column and the total number of records. We obtain the size of a column by simply dividing the two. Using this method, we determined column `l_extendedprice` of LINEITEM (size: SF $\times$ 6M tuples) has on average 5.26 byte per tuple. This is of use during estimation: for example in a scenario of SF=200 with 384 partitions this means retrieving the column from a single partition will result in $200 \times 6\text{M} \times 5.26/384 = 15.7$ MiB the model expects to be retrieved. We use the 1M-reduction **process rate** from §5.6.3 for all except Q6, which is the only query of the six which does not have a join or

reduce-by-key operation. For Q6, we set the model process rate to a value higher than any possible network rate, which means it is of no consequence as the minimum of the two rates is taken to calculate input duration in the model (§5.4.4).

5.8.3 Experimental setup

We evaluate across four scale factors $SF \in \{20, 100, 200, 500\}$ (see Tab. 5.3) for six queries (Q1, Q4, Q6, Q12, Q14, Q19). For each (SF, Q) combination, we perform runs across 49 configurations: the Cartesian product of seven number of workers and seven memory sizes (768, 1024, 1280, 1769, 2048, 2560, 4096). The seven numbers of workers are determined via the targeted number of input partitions (F) per worker: $F \in \{1, 2, 3, 4, 8, 16, 32\}$ For example, at $SF = 100$ with 192 partitions, the number of workers is $W \in \{\frac{192}{32}, \frac{192}{16}, \frac{192}{8}, \frac{192}{4}, \frac{192}{3}, \frac{192}{2}, \frac{192}{1}\} = \{6, 12, 24, 48, 64, 96, 192\}$. There is one exception for $SF = 500$, at which we do not run $F = 1$ (W=960) as it is costly and clearly not in the regime of interest. Any configuration for which any of its repeated runs results in out-of-memory or reached time limit are considered infeasible (26 out of 1134, thus we have 1108 total configurations). Each configuration is run three times, and the best data point is chosen in accordance with §5.7.1.

5.8.4 Results

The main aspect to explore is how close the recommended configuration given by the advisor is to the best configuration determined experimentally. For this, we use the

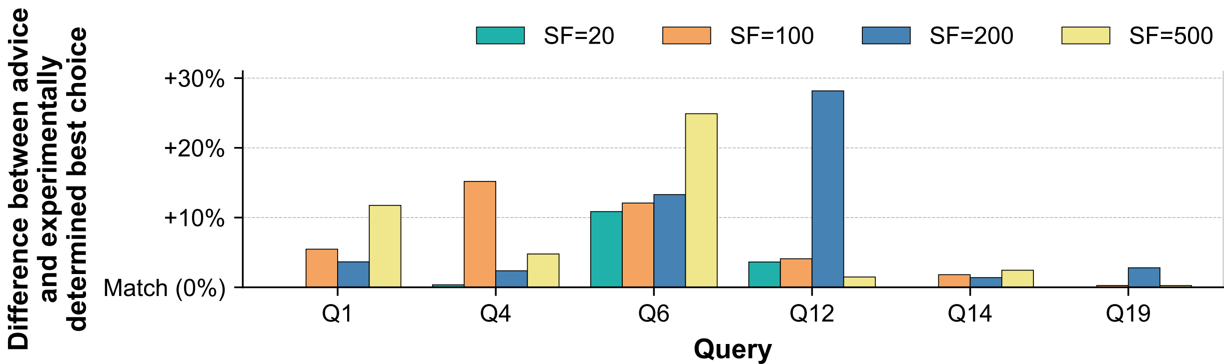
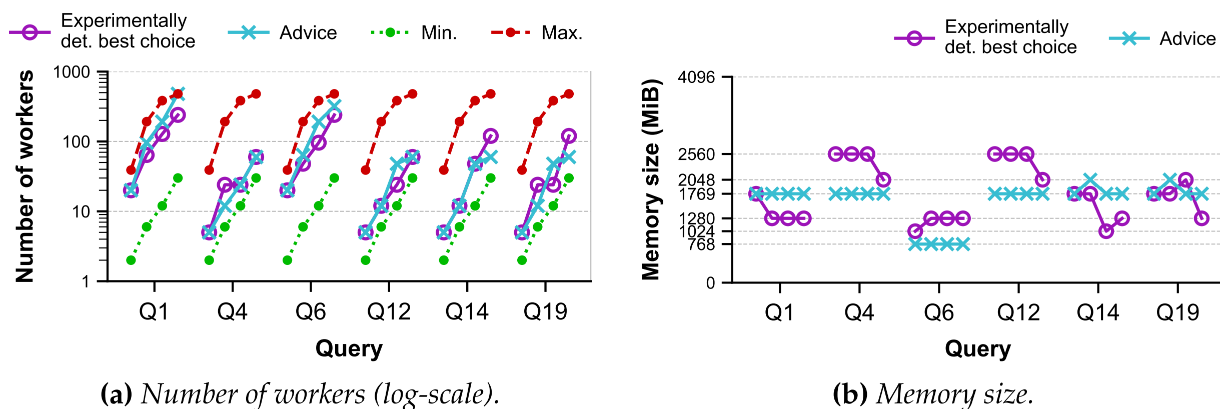


Figure 5.8: Advice outcome performance. The bar height is the difference in the distance metric value (which combines completion time and cost) between the advised configuration and the experimentally determined best configuration.



(a) *Number of workers (log-scale).*

(b) *Memory size.*

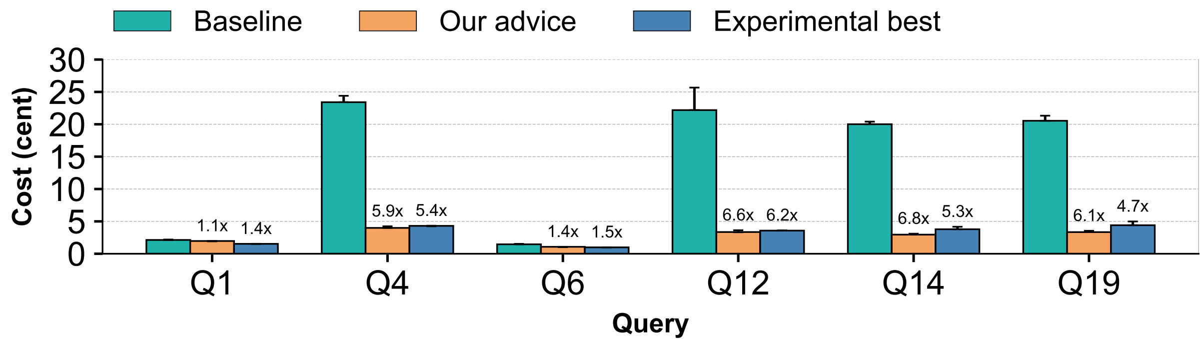
Figure 5.9: The choices made by the advisor and from the experiments. For each query, the four scale factors are plotted left to right.

metric as defined in the evaluation methodology (§5.5.2): this distance metric combines the achieved completion time and financial cost (absolute values for scale factor 500 and 100 are shown in Fig. 5.10 and Fig. 5.11 respectively), and produces a single distance value. We normalize the distance by that of the experimentally chosen best configuration distance (*i.e.,* the best observed run). We show this value for all settings in Fig. 5.8. The configurations determined best experimentally and the advisor choices are shown in Fig. 5.9. The way to read this information is as follows:

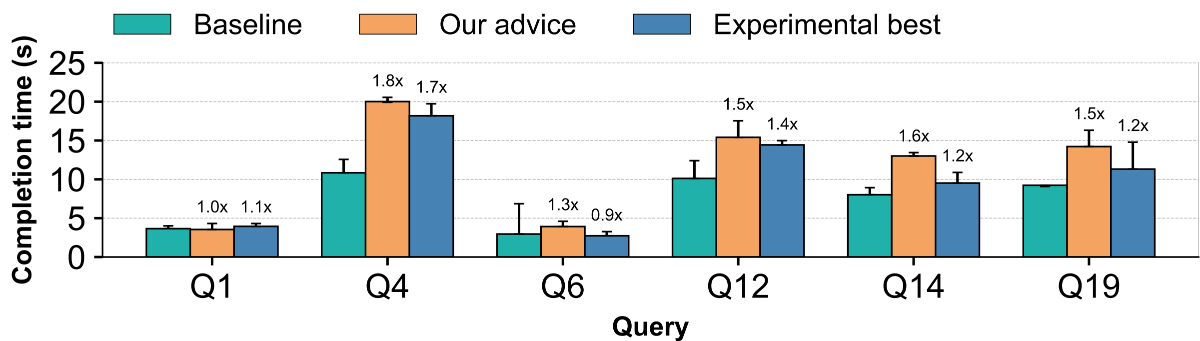
For Q14 at SF=100, the advice is the same number of workers (Fig. 5.9a) but with more memory (Fig. 5.9b) than the actual best configuration. This results in an outcome 2% worse overall (Fig. 5.8) as it finishes 2% faster but at 9% higher cost.

Although the suggested configuration not always matches the best experimental run, the advisor chooses a configuration that is within 15% off the best one determined experimentally 21 out of 24 times. This is sufficiently accurate in practice given the large amount of uncertainty inherent to cloud execution and the problem at hand (see below for the noise inherent to the experimental runs). The larger differences observed for some queries and scale factors have two main reasons.

The first reason is the inherent service variance of the cloud. Especially the start-up time of workers and network rates when reading files exhibit large fluctuations and can occasionally experience very low performance. This is influential because of (a) the completion time being in the order of seconds where even a small delay immediately causes a large delay in percentage, and (b) the dependency on the tail: the completion



(a) *Cost across the six TPC-H queries at SF=500. (ratios are times-cheaper)*

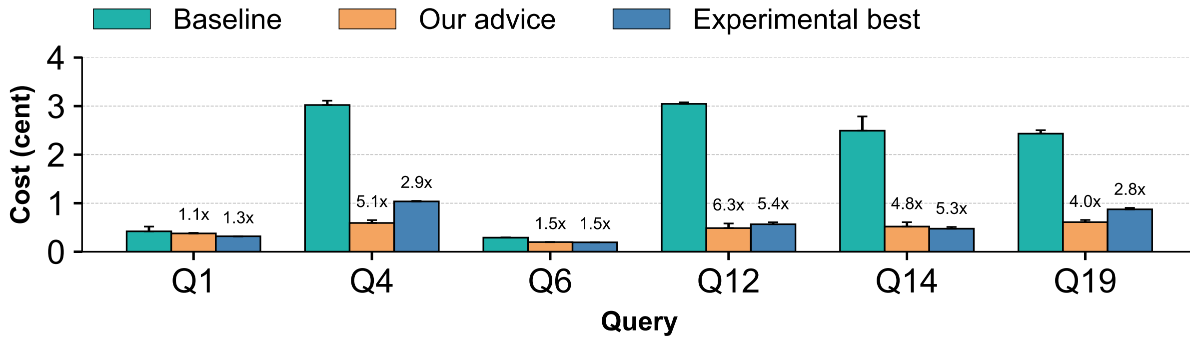


(b) *Completion time across the six TPC-H queries at SF=500. (ratios are slowdown)*

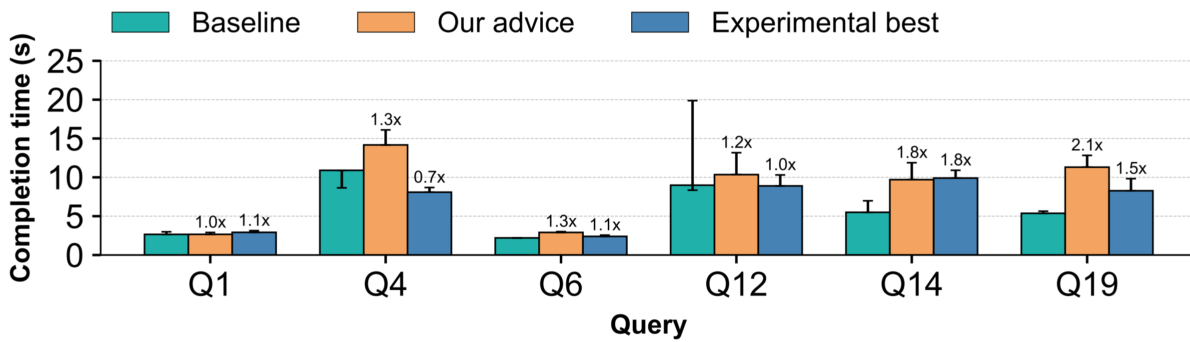
Figure 5.10: In comparison to a M=2048 MiB with F=2 baseline chosen based on [147], the advice and experimental best achieve a better tradeoff for the exchange queries in particular. (ratios above bars are relative to baseline, error bars are min-max across repetitions for the chosen configuration)

time is determined by the last worker finishing and different forms of stragglers occur often, especially at larger scales. To illustrate this effect, we compare in Fig. 5.12 the distance metric for the best and the worst of the three runs of each of the 1108 configurations (contiguous line) and the subset of kneepoints (dotted line). For the kneepoints (the points we use as reference): half have a variance of more than 10% between best and worst and almost a quarter of them have a difference of over 20%. This is why we consider that an advice that is within 15% of the best observed run is a more than reasonable approximation of a suitable configuration. In current systems, it is impossible to get more accuracy but the advice is still valuable in automating the process of deploying queries on serverless platforms.

The model limitations and assumptions are the second reason for differences between estimation and actual runs. The goal of the model is to capture the relative performance impact of the various components (*i.e.*, start-up, input, exchanges, ...). Certain aspects



(a) *Cost across the six TPC-H queries at SF=100. (ratios are times-cheaper)*



(b) *Completion time across the six TPC-H queries at SF=100. (ratios are slowdown)*

Figure 5.11: Same comparison as Fig. 5.10 for SF=100. At lower scale factors, the tradeoff gain is reduced as there are less workers involved in exchanges.

of the Lambada and AWS are difficult to estimate and are often not needed to capture the overall trend. Yet, in some cases, these can have an impact in advice outcome. In particular, issues such as the temporary bursty higher network rate when having more than one vCPU, and the effect of concurrent processing and networking on their respective rates cause unusual experimental results that are infeasible to model.

Although the first reason above is always present, the latter reason is of most interest to explain the two cases in Fig. 5.8 which are more than 20% off.

In the first case, Q6 at SF=500 has a low completion time, with the best (kneepoint) choice finishing in 2.7 s. This amplifies any delay in terms of relative outcome: the advice outcome is 3.9 s (+43%) which is far from the best run but close to the other runs. Secondly, the model overestimates network and process performance of low memory workers, and as a result prefers the cheaper lower memory size. The preference for higher number of workers stems from modeling the worker start-up as a constant (it actually grows with the number of workers but it is difficult to predict by how much, introducing

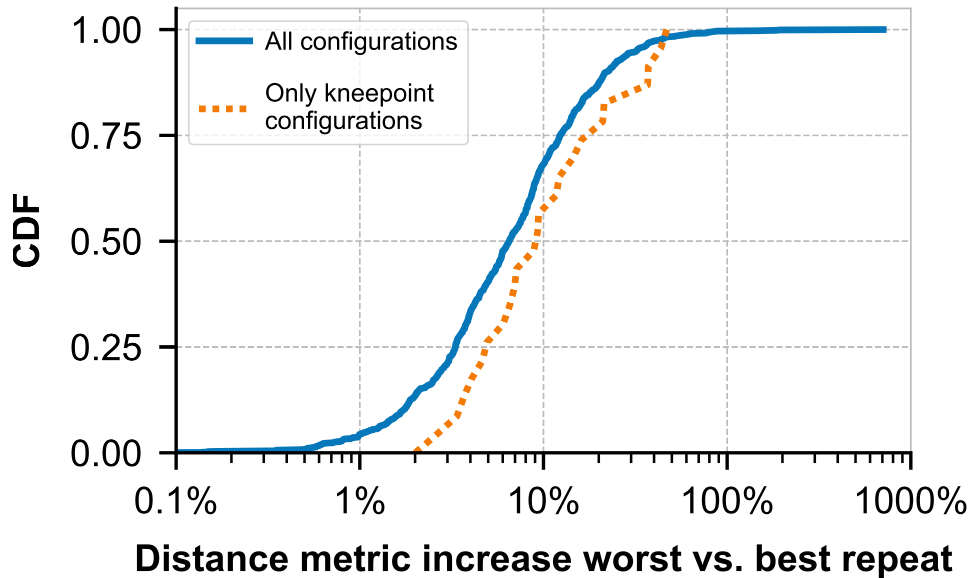


Figure 5.12: CDF comparing the performance of the worst and best repeated run of each configuration.

variances in the order of hundreds of milliseconds), and with increased probability of encountering stragglers at higher worker numbers (with stragglers being completely unpredictable). The second case, Q12 at SF=200, has an advice whose outcome is 28% off the best run with a completion time 12% slower and 41% more expensive. The model underestimates the increased processing and network rate at higher memory size, and is too optimistic regarding the additional overhead changing from a 1-level to a 2-level exchange (with 48 instead of 24 workers exceeding the threshold of 32), which experimentally does not yield a sufficiently cost-effective decrease in completion time.

Some of these aspects can be corrected an accounted for by modifying the constants or specializing the model even more for exchange heavy or scan heavy queries. We did not think it was effective to do so given the level of noise that is inherent to the system. In addition, if very many repetitions of the same query are run, a more realistic picture of its actual performance emerges but the number of runs is very high and the variance remains very high. For a system like Lambada, which targets occasional queries over cold data, the average over many runs is not relevant because queries will be run only once or twice and the probability that they run into the situations described is high as our own experiments with three runs for each configuration indicate (Figure 5.12).

Across the more representative workloads, we observe the following:

- There is significant variance in the outcome of configurations as serverless data processing systems depend on the cloud for their start-up and communication performance, yet it is still possible to give sound advice;
- The model is able to suggest a configuration with a good balance of completion time and financial cost: in the vast majority of cases it was less than 15% off the best choice a margin within the noise of the system.

5.9 Related work

There is a lot of work on resource allocation in the cloud for VMs [77] and containers [149]. Recent work from Microsoft has explored how to map serverless to the underlying VMs running functions from the perspective of the cloud provider [218] but that is a different problem from the one we address in this paper. There is also work in trying to explore the properties of serverless systems [179, 200]. Existing serverless data processing systems [147,158,201] show how to outperform alternatives in both completion and cost in select configurations, but use manually selected configurations.

Existing work on rough estimation of execution time and cost for VMs [125] does not directly translate to serverless functions due to: (a) the difference in the number of workers involved (10s for virtual machines, 100s for serverless functions), (b) the overhead incurred for data exchanges (typically through storage), and (c) the difference in task duration. Serverless operates in the few to tens of seconds range, whereas virtual machines in minutes to hours – factors insignificant at a one hour scale become so at a one second scale [179,200]. Similar to [125], we account for processing (albeit using data and rate instead of CPU hours) and network overhead, but also incorporate the factors unique to serverless query processing systems, namely startup time, exchange overhead and request cost. Another related work is Astr(e)a [97, 98], which similarly proposes a parameterized model to estimate and advise serverless configurations. However, their model focuses on long running analytic queries and thus does not incorporate startup time and exchange overhead. Their configuration advice minimizes completion time or cost provided a user-defined budget or performance constraint respectively, rather than a mechanism as ours which determines a tradeoff between completion time and cost.

In the context of Apache Spark, Sparklens [163] is a tool which estimates completion time and utilization given a certain number of compute nodes after inspecting a single run – especially useful for repeated queries but not applicable in our use case. In the context of Azure Synapse, [178] proposes the AutoExecutor framework, built upon SCOPE [40], to automatically choose the number of executors for its Spark SQL queries. TASQ [162] which similarly is built on top of SCOPE [40], trains and makes use of (graph) neural networks to optimize the amount of tokens allocated to a job. Song et al. [185] proposes a multi-objective optimization method to tune several Spark parameters including number of executors and cores, as well as memory allocation. In the work, they ran a variety of run configurations and used its resulting traces to train for each workload neural network models for various objectives including latency and cost. Besides the different target platform, these works differ from ours in the design of the model. Our goal is to provide an advisory tool with a rough estimation model based on reasoning with parameters that can be adjusted for any serverless query processing system or particular query, rather than applying learning from traces with performance metrics to determine the best allocation. Our model incorporates the most important dimensions for serverless query processing in the cloud, namely start-up, compute, network and exchange overhead. [185] similarly made use of Pareto set to determine the best configuration, although the method at which the Pareto frontier is formed (via multi-objective optimization) differs from ours (which simply estimates the entire Cartesian product of possibilities), as well as the strategy of choosing the best configuration differs as they use direct distance to the idealized fastest and cheapest point rather than using that point as normalization for distance to the origin as we do.

5.10 Summary

We have presented an advisory tool for serverless data processing designed to pick a configuration (*i.e.*, number of workers and their memory size) striking a good balance between performance (completion time) and cost. The advisory tool makes use of a rough estimation model which incorporates processing and networking, start-up, exchange overhead and request cost. We evaluated the advisor's performance using an existing serverless data processing system and show it is able to identify desirable configurations. Automated configuration facilitates the job of the user and broadens the scope at which serverless data processing is cost-effective.

6

CONCLUSION

In this final chapter we conclude the work presented in this thesis. We first provide an overall summary of both the general thesis and individual chapters in §6.1. Secondly, we consider interesting directions for future research and provide a general outlook in §6.2.

6.1 Summary

In this dissertation, we set out to make networked systems more performant, efficient, resilient and analyzable. We posit that a networked system should have access and make use of network access primitives which suit its application needs. Moreover, it is important that (application-aware) network monitoring is deployed to evaluate whether the network performance requirements are continuously being met, and diagnose (gray) failures when present in the network. We highlight the usefulness of tools and models to in their role to understand networked systems. This understanding enables both more effective and efficient provisioning, as well as support research to improve their design.

In chapter 2 we put forward the idea of bounded degradation of both partial delivery and performance as the premise for new network service primitives. In our work, we used the fair share as the basis to determine the performance and permitted loss of flows. Our proposed system achieves its guarantees through the concept of building up a budget, which grows if they run ahead of their performance goal, and depletes when needed to speed up other flows. The depletion either results in slower completion or a reduction in amount of data being delivered. We identify the fair share through the implementation of a regular probing mechanism, which requires time synchronization across participating hosts as well as warrants the congestion control protocol to converge in a timely fashion. Through packet-level simulations we evaluate the proposed mechanism. We find that

it does enable speed-up of regular flows by trading off performance with the flexible flows, especially when considering larger flows. However, due to the effect of the probing mechanism, the amount of potential speed-up is limited, and it can slow the tail completion of short flows. The key limiting factor to deployment is the need to determine its fair share (which with the probing mechanism restricts settings), as well as the limited set of scenarios in which bounded deterioration of flexible flows is desirable.

Chapter 3 characterized the concept of UNDERRADAR attacks aimed at degrading performance surreptitiously while preventing diagnosis through the use of programmable in-network devices. We identified several of such attacks, which show these programmable devices with their targeting capability are able to inhibit the flow of data and application performance as a whole with few interactions. Our work identifies several key mitigation techniques. The first recommended countermeasure is outright reducing the ability to identify key packets by restricting available information, both through the encryption of packet payload and header, as well as indirect identification by obfuscating timing and size. We note that network monitoring is most effective in mitigation if it continuously monitors all real traffic rather than depending on probing traffic or selective mirroring. These systems should be finely tuned to detect both distribution shifts which indicate gray failure and identify outliers which could have been the result of targeted interference. The second recommend countermeasure is the introduction of application-aware network monitoring, which enables the application to indicate whether its network communication requirements are being consistently met. This feedback can then be directly coupled to network-level reports, in order to identify network failures and potential UNDERRADAR attacks occurring. With this characterization and consideration of mitigations, we provide directions towards making networked systems more resilient towards such adversaries, alongside the wider benefit of improved monitoring capability.

Chapter 4 presented a network simulator for LEO satellite networks named HYPATIA. The simulator enables researchers to investigate the effect of the highly dynamic nature of LEO satellite networks on transport protocols, routing and traffic engineering. We find that the simulator is able to sufficiently scale to cover several use cases of network research interest, and is able to capture the dynamic nature at sufficient granularity. We demonstrated its utility through several experiments, which (a) show that transport protocols experience gradual and sudden delay changes and additional loss, which should be addressed in their design, and (b) brings forward the challenge to perform

routing and traffic engineering in the face of constantly shifting traffic and the viable paths they are able to take. HYPATIA constitutes a valuable effort towards enabling researcher to evaluate existing techniques and develop novel ones.

Chapter 5 introduced a parameterized approach to provision serverless query processing systems, designed to strike a good balance between completion time and financial cost. We note that exact prediction is infeasible due to inherent data-processing-related limitations such as query selectivity and data distribution, as well as due to serverless cloud platforms which provide varying opaque service quality. We design a rough estimation model, which takes into account start-up, processing, compressing, network transfer and network overheads. The model incorporates constants such as network rate, which we determine through the use of microbenchmarks. The advisor uses the model to obtain completion time and financial cost estimates for possible configurations (which we visualized in the form of a Pareto frontier), from which it picks the configuration normalized closest to the origin as the one which strikes a good balance. We evaluated the advisor using a set of six TPC-H derived queries applied to the dataset at different scales. Our automated advisor is able to pinpoint a configuration not far off the best choice in most cases. We find that there is significant variance in the actual outcome as the serverless data processing system depends on the cloud for their start-up and communication performance. Certain aspects of serverless platforms which are difficult to model additionally had an effect as the model did not incorporate them, such as temporary bursty higher network rate and the effect of concurrent processing on rates. Through automated provisioning, users are able to more efficiently make use of serverless query processing systems. By modeling serverless query processing systems, we can better understand their bottlenecks, and broaden the use case in which they are cost-effective.

6.2 Research outlook

In the upcoming sections, we discuss several interesting research directions in the diverse set of networked system topics presented in this dissertation. Lastly, we conclude with a general outlook on networked systems as a whole.

6.2.1 Improving and expanding bounded degradation primitives

Determining fair-share. The key limitation of our implementation is the determination of the fair share of flows through probing. This approach reduces its overall potential performance gain, short flows are unable to be captured, the accuracy of its fair share estimate depends on convergence rates of the transport protocol and necessitates the use of a shared timing mechanism. Rather than our approach of making use of the existing network stack and the limitations that accompany it, we could explore building a stack from the ground up which incorporates functionality to determine the fair-share (*e.g.,* this is especially promising in the data center context [101, 180, 221]). There could even be further exploration into the core definition of fairness, such that we can better define the desired outcome in cases of competition, deprioritization and degradation.

Partial delivery with deadlines as basis. We based the bounded degradation on the fair share a flow would receive. However, other bases for degradation are also conceivable. Of particular interest are deadline-aware approaches [195,203], which inherently possess a definition of what quality of service entails: whether a deadline is made. It would be interesting to expand the mapping to quality of service to not solely include whether a deadline is made, but also how much is delivered. This would expand the solution space in both the decentralized approach as well as with an centralized scheduler, as it obtains an additional degree of freedom. This can similarly be integrated in system-wide optimization like Sincronia [2], maximizing value brought to the application. It could also be generally applied to central scheduler systems such as FastPass [159], reducing the number of requested slots as appropriate.

6.2.2 Cross-layer system performance tracing

The monitoring system of the application and that of the network are often deployed to operate separately. On one side, the application-level metrics offer insight how well a networked system overall is performing, such as 99th percentile response time or throughput. On the other hand, network monitoring systems are tasked with measurements related to the network health, identifying elevated levels of congestion or loss, validating load balancing, ensuring correct routing, and pinpointing responsible problematic nodes or links. This separation of monitoring is under the premise that the two are causally related: if application-level performance degradation is experienced

and the network monitoring is indicating good operation, then the problem must be elsewhere (*e.g.*, software bugs, under-provisioning). However, an in-network programmable adversary could interfere with this relation.

An interesting future direction would be the creation of a tracing system which tracks performance from the higher application-layer abstraction of a "user request" till the drop of a packet at a particular switch. The idea is similar to that of Microsoft's 007 monitoring system's approach of performing a trace upon packet loss observation [23]. When a user request fails to meet a certain service-level objective (*e.g.*, response time is too high), a trace is initiated in which metrics from all used resources are collected and combined. The resources could for instance span network tracing and performance logs for all its connections, disk access times, queueing delay, and processing times. These traces could then be combined to reveal specific application-level targeting, and enable the identification of problematic devices. With the suspected presence of malicious devices, these cross-layer measurements should in addition be cross-checked with one another for inconsistencies. For example, [145] proposes various techniques to detect routers reporting incorrect packet forwarding behavior.

Another more direct network-level approach would be to have stringent network service guarantees (*e.g.*, deadline-based), rather than best-effort. If these network service guarantees are not met, an automated investigation can be launched in the network. The difficulty to keep track of these guarantees will depend on the network service offered, for example the coflow network service primitive in [2] will require multi-node tracking, logging and identification of causes (*e.g.*, losses).

6.2.3 Enhancing LEO network simulation further

The Hypatia simulator introduced in chapter 4 is a first step towards enabling network research for LEO satellite network simulation. It however can be improved even further in two particular directions.

Higher fidelity. In particular, it models three useful aspects of the continuously changing topology: the routing, the change of links and the varying latency of those links. There are however other considerable characteristics that can furthermore be taken into account in Hypatia, albeit they require significant development effort. To name a few: (1) more realistic modeling of radio GS-satellite communication; (2) the modeling of interference

between connections through frequency management, which would impact which ground stations and satellites can communicate and at what rate (first explorations in this direction are done in [211]); (3) the impact of physical effects on both GSLs and ISLs; or (4) routing strategies beyond shortest latency paths. These improvements would increase the fidelity of the already existing experiments (*e.g.,* congestion control behavior) as well as enable research into various other fields, such as even more network-aware adaptive routing and frequency management strategies. In addition, the fidelity of the available experiments can be improved by accounting for Earth's curvature in the maximum GSL length calculation. Another (continuously) interesting future work direction is the incorporation of the latest available constellation deployment parameterization.

Increased scalability. HYPATIA in its current form is able to satisfy several use cases of research interest. However, the addition of more modeled functionality to achieve higher fidelity, higher link capacities, larger constellations, more sophisticated routing strategies or longer simulations all will affect the perceived performance of HYPATIA. As such, investigation in how to speed up HYPATIA even further is a worthwhile pursuit. The preprocessing pipeline of HYPATIA, which calculates the routing and topology changes in advance, can be sped up by writing more optimized code or implementing it in a faster language. There is also speed-up possibility in the packet-level ns-3 simulation component by enabling parallel processing, which requires careful consideration to ensure correctness.

Of course, HYPATIA represents a single point in the space of LEO satellite networks modeling. It can be used in conjunction with other simulation, emulation and testbed tools with different modeling choices and limitations. Of particular interest is the comparison to real-world deployments to validate outcomes and further understand HYPATIA's usage and limitations. With the recent commercial introduction of LEO Internet service to the public, the creation of such testbeds has become viable [183].

6.2.4 Enhancing serverless query processing system modeling

The use case for serverless query processing is the niche of infrequent interactive analytic queries on cold data. For queries that are done frequently, or take longer than at most a few tens of seconds, alternatives such as virtual machines are likely more cost-effective and performant [147]. A model for serverless query processing as such must make

completion time estimations in the order of seconds. The estimation accuracy hinges on the predictability of startup, compute, and communication of the functions. The following would be interesting directions to improve their modeling further.

Improved QoS guarantees. The primary appeal of serverless (function-as-a-service) is that the burden of server management, load balancing and scaling is placed upon the cloud provider. Thus, by design, the infrastructure which underlies serverless functions is opaque to the user. In practice, cloud providers offer mostly best-effort rather than service-level guarantees, both in terms of start-up and serverless function configuration (*e.g.,* see Tab. 5.1). Similarly, serverless query processing systems make heavy use of cloud services, such as storage to read in their input as well as perform exchanges [147, 158], and message queueing systems (*e.g.,* to communicate with the driver). The introduction of service-level guarantees would significantly aid estimation reliability. In particular, the provision of guarantees pertaining to (a) startup rate and latency, (b) network rate and latency, and (c) cloud service guarantees (*e.g.,* with regards to file operation) is useful, as in our microbenchmarks they were observed to be most varying. Beyond cloud provider guarantees, the serverless query processing system itself can also incorporate additional techniques to improve its inherent performance reliability. Topology-aware computation [33] accounts during the execution for topological differences to achieve best performance for the targeted data processing task and its algorithmic bottlenecks. In the context of serverless, one could initially spawn a larger number of workers than intended, and quickly discard ones whose topological position would inhibit the performance. Secondly, one could add a small amount of redundant stand-by workers, which would take over for stragglers during the computation. Moreover, request success and tail response time could be improved through techniques such as hedging [52].

More customizable function configuration. Based on current cloud provider offering, in our work we limited the provisioning decision to the number of workers and their respective memory size. The other resources (compute, network, disk) are generally coupled to the chosen memory size or are constant. However, as a consequence (a) for the former, to reach the sufficiently good performance in one dimension, there is over-provisioned unused resources elsewhere (*e.g.,* computer or memory), or (b) for the latter, there is inability to increase a bottleneck resource at all (*e.g.,* network and disk). As such, increased independent customizability of the dimensions would enable the launching of functions tailored to the intended function. This would lead to a speed-up if before

the bottleneck resource dimension was constant, or improved cost-effectiveness by not having unused resource dimensions. Together, these improvements would broaden the niche use case of serverless query processing systems, and with it, the usability of the provisioning model. However, a consequence of broadening the customizability is that scheduling will involve more dimensions, which could lead to difficulty in scaling and lower levels of utilization [62]. As such, it remains to be seen if this customizability is desirable for cloud providers to make available to its users.

6.2.5 General outlook

The field of networked systems is constantly evolving to meet the requirements of increasingly challenging workloads and make use of technological developments. Not only are the existing building blocks improving (*e.g.*, faster (multi-core) processing speed, network bandwidth, and storage access rate), but also entirely new building blocks are being introduced in recent years with novel capabilities (*e.g.*, programmable switches [34], FPGAs [59], TPUs [102]). There has similarly been a broadening in the platforms and service abstractions that are offered, such as serverless [15,69,139], containers [153], QaaS [18,73], and many others. Global network infrastructure is evolving as well, with broader coverage, higher rates and lower latency through the deployment of wider spread fiber [58], better telecommunication technology (*e.g.*, 5G [143]), low Earth orbit satellite networks [28], and computing infrastructure closer to the edge [19,72,141]. Networked systems will continue to be engineered to incorporate and take advantage of these rapid developments to the fullest, enhancing their performance, resilience, analyzability, and efficiency.

BIBLIOGRAPHY

[1] Syed Hussain Abbas. Securing the network against malicious programmable switches. Master's thesis, ETH Zurich, 2020.

[2] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: near-optimal network design for coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 16–29. ACM, 2018.

[3] André Baptista Águas. Routing for a satellite mega-constellation. Master's thesis, ETH Zurich, 2020.

[4] Akamai. The state of online retail performance, 2017.

[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, 2018.

[6] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). *ACM SIGCOMM CCR*, 41(4):63–74, 2011.

[7] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, San Jose, CA, April 2012. USENIX Association.

[8] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. *ACM SIGCOMM CCR*, 43(4):435–446, 2013.

[9] Catalina Alvarez, Zhenhao He, Gustavo Alonso, and Ankit Singla. Specializing the network for scatter-gather workloads. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 267–280, 2020.

[10] Amazon. Amazon Redshift. `https://aws.amazon.com/redshift/`, 2021. Accessed: 29 November 2021.

[11] Amazon. AWS EC2: pricing. `https://aws.amazon.com/ec2/pricing/`, 2021. Accessed: 13 December 2021.

[12] Amazon. AWS Lambda: configuring lambda function options. `https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html`, 2021. Accessed: 13 December 2021.

[13] Amazon. AWS Lambda: FAQs. `https://aws.amazon.com/lambda/faqs/`, 2021. Accessed: 5 November 2021.

[14] Amazon. AWS Lambda pricing (eu-west-1, Europe, Ireland). `https://aws.amazon.com/lambda/pricing/`, 2021. Accessed: 21 October 2021.

[15] Amazon. AWS Lambda: Serverless computing. `https://aws.amazon.com/lambda/`, 2021. Accessed: 26 November 2021.

[16] Amazon. AWS S3: Cloud object storage. `https://aws.amazon.com/s3/`, 2021. Accessed: 26 November 2021.

[17] Amazon. AWS S3 pricing (eu-west-1, Europe, Ireland). `https://aws.amazon.com/s3/pricing/`, 2021. Accessed: 21 October 2021.

[18] Amazon. Amazon Athena documentation. `https://docs.aws.amazon.com/athena/`, 2022. Accessed: 27 May 2022.

[19] Amazon. AWS for the Edge. `https://aws.amazon.com/edge/`, 2022. Accessed: 27 May 2022.

[20] Apache Arrow contributors. Apache Arrow. `https://arrow.apache.org/`, 2021. Accessed: 29 October 2021.

[21] Apache Iceberg contributors. Apache Iceberg. `https://iceberg.apache.org/`, 2022. Accessed: 28 July 2022.

[22] Katerina Argyraki, Petros Maniatis, and Ankit Singla. Verifiable network-performance measurements. *ACM CoNEXT*, 2010.

[23] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hingqiang Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. *USENIX NSDI*, 2018.

[24] Astropy Collaboration, A. M. Price-Whelan, B. M. Sipőcz, H. M. Günther, P. L. Lim, S. M. Crawford, S. Conseil, D. L. Shupe, M. W. Craig, N. Dencheva, A. Ginsburg, J. T. VanderPlas, L. D. Bradley, D. Pérez-Suárez, M. de Val-Borro, T. L. Aldcroft, K. L. Cruz, T. P. Robitaille, E. J. Tollerud, C. Ardelean, T. Babej, Y. P. Bach, M. Bachetti, A. V. Bakanov, S. P. Bamford, G. Barentsen, P. Barmby, A. Baumbach, K. L. Berry, F. Biscani, M. Boquien, K. A. Bostroem, L. G. Bouma, G. B. Brammer, E. M. Bray, H. Breytenbach, H. Buddelmeijer, D. J. Burke, G. Calderone, J. L. Cano Rodríguez, M. Cara, J. V. M. Cardoso, S. Cheedella, Y. Copin, L. Corrales, D. Crichton, D. D'Avella, C. Deil, É. Depagne, J. P. Dietrich, A. Donath, M. Droettboom, N. Earl, T. Erben, S. Fabbro, L. A. Ferreira, T. Finethy, R. T. Fox, L. H. Garrison, S. L. J. Gibbons, D. A. Goldstein, R. Gommers, J. P. Greco, P. Greenfield, A. M. Groener, F. Grollier, A. Hagen, P. Hirst, D. Homeier, A. J. Horton, G. Hosseinzadeh, L. Hu, J. S. Hunkeler, Ž. Ivezić, A. Jain, T. Jenness, G. Kanarek, S. Kendrew, N. S. Kern, W. E. Kerzendorf, A. Khvalko, J. King, D. Kirkby, A. M. Kulkarni, A. Kumar, A. Lee, D. Lenz, S. P. Littlefair, Z. Ma, D. M. Macleod, M. Mastropietro, C. McCully, S. Montagnac, B. M. Morris, M. Mueller, S. J. Mumford, D. Muna, N. A. Murphy, S. Nelson, G. H. Nguyen, J. P. Ninan, M. Nöthe, S. Ogaz, S. Oh, J. K. Parejko, N. Parley, S. Pascual, R. Patil, A. A. Patil, A. L. Plunkett, J. X. Prochaska, T. Rastogi, V. Reddy Janga, J. Sabater, P. Sakurikar, M. Seifert, L. E. Sherbert, H. Sherwood-Taylor, A. Y. Shih, J. Sick, M. T. Silbiger, S. Singanamalla, L. P. Singer, P. H. Sladen, K. A. Sooley, S. Sornarajah, O. Streicher, P. Teuben, S. W. Thomas, G. R. Tremblay, J. E. H. Turner, V. Terrón, M. H. van Kerkwijk, A. de la Vega, L. L. Watkins, B. A. Weaver, J. B. Whitmore, J. Woillez, V. Zabalza, and Astropy Contributors. The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package. *The Astronomical Journal*, 156(3):123, September 2018.

[25] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Antony Rowstron. Offering network performance guarantees in multi-tenant datacenters, March 11, 2014. US Patent 8,671,407.

[26] Jeff Barr. New Amazon S3 storage class – Glacier Deep Archive. https://aws.amazon.com/blogs/aws/new-amazon-s3-storage-class-glacier-deep-archive/, 2019. Accessed: 15 August 2022.

[27] Stephen Bensley, Dave Thaler, Praveen Balasubramanian, Lars Eggert, and Glenn Judd. RFC 8257: Data center TCP (DCTCP): TCP congestion control for data centers. https://tools.ietf.org/html/rfc8257, October 2017. Accessed: 21 August 2022.

[28] Debopam Bhattacherjee, Waqar Aqeel, Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. Gearing up for the 21st century space race. In *ACM HotNets*, 2018.

[29] Debopam Bhattacherjee, Simon Kassing, Melissa Licciardello, and Ankit Singla. In-orbit computing: An outlandish thought experiment? In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 197–204, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Debopam Bhattacherjee and Ankit Singla. Network topology design at 27,000 km/hour. In *ACM CoNEXT*, 2019.

[31] Bhattacherjee, Debopam and Singla, Ankit. LEO satellite networks. https://leosatsim.github.io/, 2020. Accessed: 15 August 2022.

[32] Monowar H Bhuyan, Dhruba Kumar Bhattacharyya, and Jugal K Kalita. Network anomaly detection: methods, systems and tools. *IEEE Communications Surveys & Tutorials*, 16(1):303–336, 2014.

[33] Spyros Blanas, Paraschos Koutris, and Anastasios Sidiropoulos. Topology-aware parallel data processing: Models, algorithms and systems at scale. In *10th Annual Conference on Innovative Data Systems Research (CIDR '20)*, 2020.

[34] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3):87–95, 2014.

[35] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994.

[36] Bob Briscoe. Flow rate fairness: Dismantling a religion. *ACM SIGCOMM Computer Communication Review*, 37(2):63–74, 2007.

[37] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, nov 2019.

[38] Celestrak. Orbit visualization (e-tool per StarLink). https://celestrak.com/cesium/orbit-viz.php?tle=/NORAD/elements/supplemental/starlink.txt&satcat=/pub/satcat.txt&orbits=0&pixelSize=3&samplesPerPeriod=90&referenceFrame=1, 2020. Accessed: 15 August 2022.

[39] Cesium. CesiumJS. https://cesium.com/cesiumjs/, 2020. Accessed: 15 August 2022.

[40] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, aug 2008.

[41] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with Karuna. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 174–187. ACM, 2016.

[42] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. *ACM HotNets*, pages 31–36, 2012.

[43] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[44] Cisco. Cisco security advisories and alerts. https://tools.cisco.com/security/center/publicationListing.x?product=Cisco. Accessed: 8 September 2018.

[45] Cisco. Implementing quality of service policies with DSCP. https://www.cisco.com/c/en/us/support/docs/quality-of-service-qos/qos-

packet-marking/10103-dscpvalues.html, 2008. Accessed: 15 August 2022.

[46] Cisco. Cisco Nexus 3000 series and 3500 platform switches insecure default credentials vulnerability. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20160302-n3k, March 2016. Accessed: 8 September 2018.

[47] Cisco. Cisco Prime data center network manager debug remote code execution vulnerability. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20170607-dcnm1, June 2017. Accessed: 13 September 2018.

[48] Cisco. QoS: Congestion management configuration guide, Cisco IOS XE release 3s. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/qos_conmgt/configuration/xe-3s/qos-conmgt-xe-3s-book/qos-conmgt-oview.html, 2018. Accessed: 15 August 2022.

[49] Benoit Claise. Cisco systems NetFlow services export version 9, 2004.

[50] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The Snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 215–226, New York, NY, USA, 2016. Association for Computing Machinery.

[51] Databricks. Databricks: The data and AI company. https://databricks.com/, 2021. Accessed: 26 November 2021.

[52] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[53] Inigo del Portillo, Bruce G Cameron, and Edward F Crawley. A technical comparison of three low Earth orbit satellite constellation systems to provide global broadband. In *Acta Astronautica*. Elsevier, 2019.

[54] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. PCC: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, 2015.

[55] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[56] Elias Eccli. Starlink constellation animation (2019/11/17 - 2021/02/27). `https://www.youtube.com/watch?v=rddTXl_7Wr8`, 2021. Accessed: 15 August 2022.

[57] Wesley Eddy. RFC 4987: TCP SYN flooding attacks and common mitigations. `https://tools.ietf.org/html/rfc4987`, August 2007. Accessed: 15 August 2022.

[58] FTTH Council Europe. Fibre market panorama 2021 press release. `https://www.ftthcouncil.eu/Portals/1/Fibre_Market_Panorama_2021_PR_def.pdf`, 2021. Accessed: 27 May 2022.

[59] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure Accelerated Networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[60] Hanjing Gao. Making machine learning friendlier in the cloud. Master's thesis, ETH Zurich, 2020.

[61] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 1. ACM, 2015.

[62] Pedro García-López, Marc Sánchez-Artigas, Simon Shillaker, Peter Pietzuch, David Breitgand, Gil Vernik, Pierre Sutra, Tristan Tarrant, and Ana Juan Ferrer.

ServerMix: Tradeoffs and challenges of serverless data analytics. *arXiv preprint arXiv:1907.11465*, 2019.

[63] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, 2018.

[64] GeoPy contributors. GeoPy: Geocoding library for Python. `https://github.com/geopy/geopy`, 2020. Accessed: 15 August 2022.

[65] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of TCP. *ACM SOSR*, pages 61–74, 2017.

[66] Giacomo Giuliari, Tobias Klenze, Markus Legner, David Basin, Adrian Perrig, and Ankit Singla. Internet backbones in space. In *ACM SIGCOMM CCR*, 2020.

[67] Google. Survey report: Behind the growing confidence in cloud security, 2019.

[68] Google. Cloud Functions execution environment: File system. `https://cloud.google.com/functions/docs/concepts/exec#file_system`, 2021. Accessed: 13 December 2021.

[69] Google. Google Cloud: Cloud Functions. `https://cloud.google.com/functions`, 2021. Accessed: 26 November 2021.

[70] Google. Google Cloud Functions pricing. `https://cloud.google.com/functions/pricing`, 2021. Accessed: 13 December 2021.

[71] Google. Google Cloud: VM instance pricing. `https://cloud.google.com/compute/vm-instance-pricing`, 2021. Accessed: 13 December 2021.

[72] Google. Google Distributed Cloud. `https://cloud.google.com/distributed-cloud`, 2021. Accessed: 27 May 2022.

[73] Google. BigQuery. `https://cloud.google.com/bigquery`, 2022. Accessed: 27 May 2022.

[74] Loren Grush. Amazon's Project Kuiper books up to 83 rockets to launch its Internet-beaming satellites. `https://www.theverge.com/2022/4/5/2301`

`0245/amazon-project-kuiper-megaconstellation-arianespace-ula-blue-origin`, 2022. Accessed: 18 April 2022.

[75] Nicolas Guilbaud and Ross Cartlidge. Localizing packet loss in a large complex network. *NANOG 57*, 2013.

[76] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. *ACM SIGCOMM CCR*, 45(4):139–152, 2015.

[77] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, SOSP '21, 2020.

[78] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[79] Mark Handley. Delay is not an option: Low latency routing in space. In *ACM HotNets*, 2018.

[80] Mark Handley. Starlink revisions, Nov 2018. `https://www.youtube.com/watch?v=QEIUdMiColU`, 2018. Accessed: 15 August 2022.

[81] Mark Handley. Using ground relays for low-latency wide-area routing in megaconstellations. In *ACM HotNets*, 2019.

[82] Jianwei Hao, Ting Jiang, Wei Wang, and In Kee Kim. An empirical analysis of VM startup times in public IaaS clouds. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 398–403, 2021.

[83] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren

Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[84] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[85] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *Conference on Innovative Data Systems Research, CIDR'19*, 2019.

[86] Thomas Henderson and Randy Katz. Network simulation for LEO satellite networks. In *18th International Communications Satellite Systems Conference and Exhibit*. AIAA, 2000.

[87] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. RFC 6582: The NewReno modification to TCP's fast recovery algorithm, April 2012. Accessed: 19 August 2022.

[88] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[89] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 15–26, 2013.

[90] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. *ACM HotOS*, pages 150–155, 2017.

[91] HughesNet. HughesNet: High-speed satellite Internet. `https://www.hughesnet.com/`, 2020. Accessed: 15 August 2022.

[92] Iridium Communications Inc. Iridium NEXT. https://web.archive.org/web/20200718205415/https://www.iridiumnext.com/, 2020. Accessed: 15 August 2022.

[93] Iridium Communications Inc. Iridium satellite communications. https://www.iridium.com/, 2020. Accessed: 15 August 2022.

[94] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined WAN. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[95] Virajith Jalaparti. *Improving the end-to-end latency of datacenter applications using coordination across application components*. PhD thesis, University of Illinois at Urbana-Champaign, 2015.

[96] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. *ACM SIGCOMM Computer Communication Review*, 43(4):219–230, 2013.

[97] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. Astra: Autonomous serverless analytics with cost-efficiency and QoS-awareness. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 756–765. IEEE, 2021.

[98] Jananie Jarachanthan, Li Chen, Fei Xu, and Bo Li. Astrea: Auto-serverless analytics towards cost-efficiency and QoS-awareness. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[99] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[100] Ryan Jones. Top 5 physical ways into a data center, May 2010. Presented at YSTS 4.0.

[101] Lavanya Jose, Lisa Yan, Mohammad Alizadeh, George Varghese, Nick McKeown, and Sachin Katti. High speed networks need proactive congestion control. In

Proceedings of the 14th ACM Workshop on Hot Topics in Networks, page 14. ACM, 2015.

[102] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[103] Karthikeyan C Kasiviswanathan. Postmortem of a compromised MikroTik router. `https://www.symantec.com/blogs/threat-intelligence/hacked-mikrotik-router`, August 2018. Accessed: 13 September 2018.

[104] Simon Kassing, Hussain Abbas, and Hanjing Gao. basic-sim: ns-3 module to make experimental simulation of networks a bit easier. `https://github.com/snkas/basic-sim`, 2021. Accessed: 15 August 2022.

[105] Simon Kassing, Hussain Abbas, Laurent Vanbever, and Ankit Singla. Order P4-66: Characterizing and mitigating surreptitious programmable network device exploitation. *arXiv preprint arXiv:2103.16437*, 2021.

[106] Simon Kassing, Debopam Bhattacherjee, André Baptista Águas, Jens Eirik Saethre, and Ankit Singla. Exploring the "Internet from space" with Hypatia. In *Proceedings of the ACM Internet Measurement Conference*, IMC '20, page 214–229, New York, NY, USA, 2020. Association for Computing Machinery.

[107] Simon Kassing, Vojislav Dukic, Ce Zhang, and Ankit Singla. New primitives for bounded degradation in network service. *arXiv preprint arXiv:2208.08429*, 2022.

[108] Simon Kassing, Ingo Müller, and Gustavo Alonso. Resource allocation in serverless query processing. *arXiv preprint arXiv:2208.09519*, 2022.

[109] Simon Kassing and Ankit Singla. Codebind: tying networking papers to their experiment code, 2021.

[110] Simon Kassing, Asaf Valadarsky, Gal Shahaf, Michael Schapira, and Ankit Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 281–294, New York, NY, USA, 2017. Association for Computing Machinery.

[111] Amiko Kauderer and Kim Dismukes. NASA HSF: Definition of two-line element set coordinate system. `https://spaceflight.nasa.gov/realdata/sightings/SSapplications/Post/JavaSSOP/SSOP_Help/tle_def.html`, 2011. Accessed: 9 June 2020.

[112] Youngbin Kim and Jimmy Lin. Serverless data analytics with Flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 451–455. IEEE, 2018.

[113] Tobias Klenze, Giacomo Giuliari, Christos Pappas, Adrian Perrig, and David Basin. Networking in heaven as on Earth. In *ACM HotNets*, 2018.

[114] Nicole Kobie. Nobody is safe from Russia's colossal hacking operation. `https://www.wired.co.uk/article/russia-hacking-russian-hackers-routers-ncsc-uk-us-2018-syria`, April 2018. Accessed: 12 September 2018.

[115] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C Snoeren. Detection and localization of network black holes. In *IEEE INFOCOM 2007-26th IEEE International Conference on Computer Communications*, pages 2180–2188. IEEE, 2007.

[116] Mark Krebs. Satellite constellation, May 9, 2017. US Patent 9,647,749.

[117] M. Kuehlewind, G. Hazel, S. Shalunov, and J. Iyengar. RFC 6817: Low extra delay background transport (LEDBAT). `https://tools.ietf.org/html/rfc6817`, December 2012. Accessed: 15 August 2022.

[118] Kuiper Systems LLC. Application of Kuiper Systems LLC for authority to launch and operate a non-geostationary satellite orbit system in Ka-band frequencies. `https://licensing.fcc.gov/myibfs/download.do?attachment_key=1773885`, 2019. File Number: SAT-LOA-20190704-00057. Accessed: 15 August 2022.

[119] Kuiper Systems LLC. USASAT-NGSO-8A ITU filing. `https://www.itu.int/ITU-R/space/asreceived/Publication/DisplayPublication/8716`, March 2019. Accessed: 15 August 2022.

[120] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering: The road not taken. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 157–170, Renton, WA, April 2018. USENIX Association.

[121] Aleksandar Kuzmanovic and Edward W Knightly. Low-rate TCP-targeted denial of service attacks: the shrew vs. the mice and elephants. *ACM SIGCOMM*, 2003.

[122] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, New York, NY, USA, 2010. Association for Computing Machinery.

[123] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467. ACM, 2016.

[124] Dennis Legezo. LuckyMouse hits national data center to organize country-level waterholing campaign. `https://securelist.com/luckymouse-hits-national-data-center/86083/`, June 2018. Kaspersky lab. Accessed: 12 September 2018.

[125] Viktor Leis and Maximilian Kuschewski. Towards cost-optimal query processing in the cloud. *Proceedings of the VLDB Endowment*, 14(9):1606–1612, 2021.

[126] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. *USENIX NSDI*, 2016.

[127] Linux contributors. /proc/sys/net/ipv4/* variables. `https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt`. Accessed: 8 September 2018.

[128] Linux contributors. The Linux PTP project. `http://linuxptp.sourceforge.net/`, 2019. Accessed: 15 August 2022.

[129] Ke Liu, Shin-Yeh Tsai, and Yiying Zhang. ATP: a datacenter approximate transmission protocol. *arXiv preprint arXiv:1901.01632*, 2019.

[130] Glenn Lopez, Daniel T Seaton, Andrew Ang, Dustin Tingley, and Isaac Chuang. Google BigQuery for education: Framework for parsing and analyzing edX MOOC data. In *Proceedings of the fourth (2017) ACM conference on learning@ scale*, pages 181–184, 2017.

[131] Samuel Madden, Jialin Ding, Tim Kraska, Sivaprasad Sudhir, David E Cohen, Timothy Mattson, and Nesime Tatbul. Self-organizing data containers. In *Conference on Innovative Data Systems Research (CIDR'22)*, 2022.

[132] Adam Mann, Tereza Pultarova, and Elizabeth Howell. Starlink: SpaceX's satellite Internet project. `https://www.space.com/spacex-starlink-satellites.html`, 2022. Accessed: 18 April 2022.

[133] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210. ACM, 2017.

[134] Met Office. *Cartopy: a cartographic python library with a matplotlib interface*. Exeter, Devon, 2010 - 2015.

[135] Microsoft. Azure Container Instances pricing. `https://azure.microsoft.com/en-in/pricing/details/container-instances/`, 2021. Accessed: 13 December 2021.

[136] Microsoft. Azure Functions scale and hosting. `https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale`, 2021. Accessed: 13 December 2021.

[137] Microsoft. Azure Linux virtual machines pricing. `https://azure.microsoft.com/en-in/pricing/details/virtual-machines/linux/`, 2021. Accessed: 13 December 2021.

[138] Microsoft. Data Lake: Microsoft Azure. `https://azure.microsoft.com/en-us/solutions/data-lake/`, 2021. Accessed: 26 November 2021.

[139] Microsoft. Microsoft Azure Functions: Serverless apps and computing. `https://azure.microsoft.com/en-us/services/functions/`, 2021. Accessed: 26 November 2021.

[140] Microsoft. Project Kudu wiki – understanding the Azure App Service file system: Temporary files. https://github.com/projectkudu/kudu/wiki/Understanding-the-Azure-App-Service-file-system#temporary-files, 2021. Accessed: 26 January 2022.

[141] Microsoft. Azure Stack Edge. https://azure.microsoft.com/en-us/products/azure-stack/edge/, 2022. Accessed: 27 May 2022.

[142] Mininet contributors. Mininet: An instant virtual network on your laptop (or other PC). http://mininet.org/, 2018. Accessed: 5 June 2018.

[143] Rupendra Nath Mitra and Dharma P Agrawal. 5G mobile technology: A survey. *ICT Express*, 1(3):132–137, 2015.

[144] Alper T Mizrak, Stefan Savage, and Keith Marzullo. Detecting malicious packet losses. *IEEE TPDS*, 20(2):191–206, 2009.

[145] Alper Tugay Mizrak, Yu-Chung Cheng, Keith Marzullo, and Stefan Savage. Detecting and isolating malicious routers. *IEEE TDSC*, 3(3):230–244, 2006.

[146] Jeff Mogul and Jitu Padhye. HotNets-XVI Dialogue: in-network computation is a dumb idea whose time has come. https://conferences.sigcomm.org/hotnets/2017/dialogues/dialogue140.pdf, 2017. Accessed: 15 August 2022.

[147] Ingo Müller, Renato Marroquín, and Gustavo Alonso. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 115–130, New York, NY, USA, 2020. Association for Computing Machinery.

[148] Geetha Nandikotkur. 200,000 Cisco network switches reportedly hacked. https://www.bankinfosecurity.com/200000-cisco-network-switches-reportedly-hacked-a-10788, April 2018. Accessed: 12 September 2018.

[149] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with POP. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, 2021.

[150] NASA. Design and integration tools: General mission analysis tool (GMAT). https://software.nasa.gov/software/GSC-17177-1, 2020. Accessed: 15 August 2022.

[151] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at Facebook. *USENIX NSDI*, 13:385–398, 2013.

[152] ns-3 contributors. ns-3 network simulator. https://www.nsnam.org/, 2020. Accessed: 15 August 2022.

[153] Open Container Initiative contributors. Open Container Initiative. https://opencontainers.org/, 2022. Accessed: 27 May 2022.

[154] P4lang bmv2 contributors. Behavioral model (bmv2). https://github.com/p4lang/behavioral-model, 2018. Accessed: 5 June 2018.

[155] P4lang tutorials contributors. P4 language tutorials. https://github.com/p4lang/tutorials, 2018. Accessed: 5 June 2018.

[156] Sonia Panchen, Peter Phaal, and Neil McKee. RFC 3176: InMon Corporation's sFlow: A method for monitoring traffic in switched and routed networks. https://tools.ietf.org/html/rfc3176, September 2001. Accessed: 21 August 2022.

[157] Pedro Silva. ns3-satellite. https://gitlab.inesctec.pt/pmms/ns3-satellite, 2016. Accessed: 15 August 2022.

[158] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 131–141, New York, NY, USA, 2020. Association for Computing Machinery.

[159] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. *ACM SIGCOMM Computer Communication Review*, 44(4):307–318, 2015.

[160] Peter Phaal. sFlow sampling rates. https://blog.sflow.com/2009/06/sampling-rates.html, June 2009. Accessed: 10 September 2018.

[161] Gregory Pickett. Staying persistent in software defined networks. *DefCon*, 2015.

[162] Anish Pimpley, Shuo Li, Anubha Srivastava, Vishal Rohra, Yi Zhu, Soundararajan Srinivasan, Alekh Jindal, Hiren Patel, Shi Qiao, and Rathijit Sen. Optimal resource allocation for serverless queries, 2021.

[163] Qubole. Qubole Sparklens tool for performance tuning Apache Spark. `https://github.com/qubole/sparklens`, 2022. Accessed: 27 January 2022.

[164] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S Pai, and Michael J Freedman. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 275–288, 2014.

[165] Benjamin Ransford and Luis Ceze. SAP: an architecture for selectively approximate wireless communication, 2015.

[166] Brandon Rhodes and PyEphem contributors. PyEphem: Scientific-grade astronomy routines for Python. `https://rhodesmill.org/pyephem/`, 2020. Accessed: 15 August 2022.

[167] Brandon Rhodes and SGP4 contributors. SGP4: Python version of the SGP4 satellite position library. `https://github.com/brandon-rhodes/python-sgp4`, 2020. Accessed: 15 August 2022.

[168] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. *USENIX NSDI*, pages 595–612, 2017.

[169] Piotr Rygielski and Samuel Kounev. Network virtualization for QoS-aware resource management in cloud data centers: A survey. *PIK-Praxis der Informationsverarbeitung und Kommunikation*, 36(1):55–64, 2013.

[170] Jens Eirik Saethre. Simulations of satellite-based low-latency Internet. Master's thesis, ETH Zurich, 2019.

[171] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM TOCS*, 2(4):277–288, 1984.

[172] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. *ACM HotNets*, 2017.

[173] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[174] Kazunori Sato. An inside look at Google BigQuery. White paper: `https://cloud.google.com/files/BigQueryTechnicalWP.pdf`, 2012. Accessed: 26 March 2022.

[175] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. *ACM SIGCOMM CCR*, 29(5):71–78, 1999.

[176] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM*, 64(5), apr 2021.

[177] Skipper Seabold and Josef Perktold. Statsmodels: Econometric and statistical modeling with Python. In *9th Python in Science Conference*, 2010.

[178] Rathijit Sen, Abhishek Roy, and Alekh Jindal. Predictive price-performance optimization for serverless query processing. *CoRR*, abs/2112.08572, 2021.

[179] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, 2020.

[180] Naveen Kr Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 67–82, 2017.

[181] Z. Shelby, C. Bormann, and K. Hartke. RFC 7252: The constrained application protocol (CoAP). https://tools.ietf.org/html/rfc7252, June 2014. Accessed: 15 August 2022.

[182] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A decade of Clos topologies and centralized control in Google's datacenter network. *ACM SIGCOMM*, 2015.

[183] Ankit Singla. SatNetLab: a call to arms for the next global Internet testbed. *ACM SIGCOMM Computer Communication Review*, 51(2):28–30, 2021.

[184] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[185] Fei Song, Khaled Zaouk, Chenghao Lyu, Arnab Sinha, Qi Fan, Yanlei Diao, and Prashant Shenoy. Boosting cloud data analytics using multi-objective optimization, 2020.

[186] SpaceX. Application for approval for orbital deployment and operating authority for the SpaceX NGSO satellite system. https://licensing.fcc.gov/myibfs/download.do?attachment_key=1158349, 2016. File Number: SAT-LOA-20161115-00118. Accessed: 21 August 2022.

[187] SpaceX. SpaceX non-geostationary satellite system. https://licensing.fcc.gov/myibfs/download.do?attachment_key=1877671, 2019. File Number: SAT-MOD-20190830-00087. Accessed: 15 August 2022.

[188] SpaceX. Starlink. https://www.starlink.com/, 2020. Accessed: 15 August 2022.

[189] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active device and link failure localization in data center networks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 599–614, 2019.

[190] Telesat. Telesat: Global satellite operators. https://www.telesat.com/, 2020. Accessed: 15 August 2022.

[191] Telesat. Telesat low Earth orbit non-geostationary satellite system. Technical information supplement to schedule S. https://licensing.fcc.gov/myibfs/download.do?attachment_key=2378320, 2020. File Number: SAT-MPL-20200526-00053. Accessed: 15 August 2022.

[192] TPC. TPC-H. http://tpc.org/tpc_documents_current_versions/current_specifications5.asp, 2021. Accessed: 29 October 2021.

[193] TPC. TPC-H dbgen tool. http://tpc.org/tpc_documents_current_versions/current_specifications5.asp, 2021. Accessed: 29 October 2021.

[194] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is advance knowledge of flow sizes a plausible assumption? In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 565–580, 2019.

[195] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter TCP (D2TCP). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[196] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 407–420, 2017.

[197] Viasat. Viasat: Global communications | services, solutions & satellite Internet. https://www.viasat.com/, 2020. Accessed: 15 August 2022.

[198] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, 2021.

[199] Jason Min Wang, Ying Wang, Xiangming Dai, and Brahim Bensaou. SDN-based multi-class QoS-guaranteed inter-data center traffic management. In *2014 IEEE*

3rd International Conference on Cloud Networking (CloudNet), pages 401–406. IEEE, 2014.

[200] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, 2018.

[201] Mike Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. Boxer: Data analytics on network-enabled serverless platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR'21)*, pages 1–8, Chaminade, USA, 2021. ETH Zurich.

[202] Thomas Williams, Colin Kelley, and Gnuplot contributors. Gnuplot: a portable, multi-platform, command-line driven graphing utility. `https://sourceforge.net/projects/gnuplot/`, 2020. Accessed: 15 August 2022.

[203] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):50–61, aug 2011.

[204] Lloyd Wood. SaVi: satellite constellation visualization. *First Annual CCSR Research Symposium (CRS 2011)*, 2011.

[205] Lloyd Wood, Patrick Worfolk, et al. SaVi - satellite constellation visualization software. `https://savi.sourceforge.io/`, 2017. Accessed: 15 August 2022.

[206] Jiacheng Xia, Gaoxiong Zeng, Junxue Zhang, Weiyan Wang, Wei Bai, Junchen Jiang, and Kai Chen. Rethinking transport layer design for distributed machine learning. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 22–28. ACM, 2019.

[207] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for Internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, 2018.

[208] BS Yeo. An average end-to-end packet delay analysis for LEO satellite networks. In *Proceedings IEEE 56th Vehicular Technology Conference*, volume 4, pages 2012–2016. IEEE, 2002.

[209] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over HTTP. *SIGCOMM Comput. Commun. Rev.*, 45(4):325–338, aug 2015.

[210] Chen Yu, Hanlin Tang, Cedric Renggli, Simon Kassing, Ankit Singla, Dan Alistarh, Ce Zhang, and Ji Liu. Distributed learning over unreliable networks. In *International Conference on Machine Learning*, pages 7202–7212, 2019.

[211] Fizza Zafar. Analyzing the impact of GEO arc avoidance on LEO constellation performance. Master's thesis, ETH Zurich, 2021.

[212] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[213] Kim Zetter. NSA laughs at PCs, prefers hacking routers and switches. `https://www.wired.com/2013/09/nsa-router-hacking/`, September 2013. Accessed: 12 September 2018.

[214] Kim Zetter. Unpatched routers being used to build vast proxy army, spy on networks. `https://arstechnica.com/information-technology/2018/09/unpatched-routers-being-used-to-build-vast-proxy-army-spy-on-networks/`, September 2018. Accessed: 13 September 2018.

[215] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. AWStream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 236–252. ACM, 2018.

[216] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. CODA: Toward automatically identifying and scheduling coflows in the dark. *ACM SIGCOMM*, pages 160–173, 2016.

[217] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. *ACM IMC*, pages 78–85, 2017.

[218] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, 2021.

[219] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. *ACM SIGCOMM CCR*, 45(4):479–491, 2015.

[220] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 362–375, 2017.

[221] Danyang Zhuo, Qiao Zhang, Vincent Liu, Arvind Krishnamurthy, and Thomas Anderson. Rack-level congestion control. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 148–154. ACM, 2016.