Improving Application Connectivity with Network-Conscious Systems

by

HyunJong Lee

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2022

Doctoral Committee:

Professor Brian D. Noble, Chair
Professor Vineet R. Kamat
Associate Professor Harsha V. Madhyastha
Professor Kang G. Shin

HyunJong Lee
hyunjong@umich.edu
ORCID iD: 0000-0003-4261-2809

To my parents.

그동안 고생하신 아버지와 어머니께 이글을 바칩니다.

ACKNOWLEDGEMENTS

This dissertation has a single author, but it was made possible by many people whom I truly appreciate. First of all, I am in eternal debt to my adviser, Professor Brian Noble. Since Summer 2019, he has nurtured me to grow not only in research but also personally. Ranging from the COVID lock-down to switching career paths, Brian has been always supportive and truly caring. There is no doubt that I could not have completed my Ph.D. without Brian's unconditional support and help. Secondly, I would like to thank to my former adviser Professor Jason Flinn. He taught me valuable skills in designing systems in my first three years of graduate school. Also, he generously offered me a support to move to Michigan, even though we just met once in SOSP 2015, right after Karsten's unexpected death. Thirdly, I would like to thank my former former adviser Professor Karsten Schwan for convincing me to pursue my Ph.D. after my undergraduate study. I would also like to acknowledge the members of my thesis committee members, Professor Kang G. Shin, Harsha V. Madhyastha, and Vineet R. Kamat for providing valuable feedback on my dissertation.

I would like to thank Landon P. Cox and Shadi A. Noghabi for mentoring me throughout the BumbleBee project after my Microsoft Research internship. Landon and Shadi taught me practical research skills in identifying and solving core problems in systems. I would also like to thank Hee Won Lee for helping me to continue the Croesus project after my AT&T Research internship and Samsung Electronics internship. I thank Minsung Jang for mentoring me since undergraduate study.

I also thank members of BBB 4929. I have wonderful memories with Vaspol

Ruamviboonsuk, Chris J. Baik, Xianzheng Dou, Muhammed Uluyol, Ayush Goel, Jingyuan Zhu, David Devecsery, Kyungmin (little Jason) Lee, and Mike Chow. Vaspol, Chris, and Xianzheng always cheered me up when I was in a bad mood after a research meeting. Ayush and Jingyuan helped me to organize office drink nights while Muhammed involuntarily became designated driver. David taught me how to be humble. Kyungmin passed tips on to me about working with Jason and Brian. Mike left an office retreat legacy with an invitation template.

I also want to thank my roommates, Henry Hyunsik Moon and Hardik Parwana. Henry has supported me in various aspects of my life. Specifically, during the RAVEN project, he generously drove in many nights to conduct live experiments for several hours. Hardik became my good friend by cheering me up when I was having a difficult time with job and personal obstacles. I would like to thank my fiancée Angelene Minji Ku for supporting me since 2019. Lastly but most importantly, I would like to thank my parents. They have provided me long, unconditional support in every aspects of my life, especially since leaving Korea in 2006. They supported every either good or bad decisions I have made. Without my parents, certainly, I could have not been where I am today.

My Ph.D. took seven years and three advisers–Brian Noble, Jason Flinn, and Karsten Schwan. It came with many downfalls: Karsten passing away, Jason moving to Facebook, endless paper rejections, and the COVID-19 lockdown. My curiosity toward knowledge is one facet of motivation; my eagerness to prove that even with so many downfalls one can complete a Ph.D. is another deep motivation and helped me to successfully reach the final goal.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Application connectivity is ubiquitous and essential: sharing an opinion on social media, live video-streaming a historic moment, warning a driver about hazardous road conditions, etc. Unfortunately, application connectivity suffers from *turbulent* network environments, because existing underlying systems are largely *oblivious* to network turbulence, resulting in poor application performance and degraded user experiences.

This dissertation presents basic building blocks toward building *network-conscious* software systems to improve application connectivity. It demonstrates how applications suffer from turbulent networking environments through series of measurement studies. Based on lessons learned in the studies, it introduces three systems. The first system employs redundancy in high-mobility environments to hedge against uncertainty in future connectivity performance. The second is a practical migration system that co-locates computation closer to end-devices. The third system allows application developers to inject lightweight scripts in-network to perform application-aware adaptations closer to the source of turbulence. Collectively, these observations and systems show the value of network-conscious scheduling, migration, and adaptation.

CHAPTER I

Introduction

Connected applications play a critical role in today's society: sharing an opinion on social media, video-streaming a historic moment, or warning a driver about hazardous road conditions. Unfortunately, existing connected applications are largely *oblivious* to turbulent network connectivity. Applications routinely suffer from such turbulence, resulting in poor application performance and a compromised user-experience. If such applications were *conscious* of underlying network turbulence, they would provide better performance and functionality.

1.1 Motivation

Increasingly, end-devices and their applications are connected (e.g., vehicles, phones, drones). Gartner, a research and advisory firm, reports that in 2019 alone, 4.81 billion new connected devices were deployed [46]. In the same year, nearly 50% of vehicles on the road in the U.S. already had embedded connectivity over cellular networks [29]. Cisco predicts that traffic from wireless and mobile devices combined will account for 71% of the total Internet traffic by the end of 2022 [68].

Applications that run on these connected devices rely on that connectivity to leverage elastic computing resources in the infrastructure at the edge and in the cloud. Such resources provide significantly more capability than is possible on the connected

devices alone. For example, Apple's Siri [5] can perform a handful of simple operations when disconnected (e.g., dialing a number). When connected, it becomes a powerful tool with a rich set of operations (e.g., video recommendation from Twitter trends), provided in part by the more capable remote portion of the service. Having ready, reliable access to such back-end services is critical for the full functioning of many connected applications and their users.

Unfortunately, today's network remains highly *turbulent*. Applications on connected devices experience tail latency [130, 79], bandwidth hiccups [150, 95], packet loss [131], and even disconnection [119]. Applications do their best to reactively adapt to volatile network environments at the end-points. Siri having limited operations when disconnected is an example. Likewise, video streaming services commonly use adaptive bit-rate (ABR) technologies such as MPEG-DASH [145] and HLS [114] to deliver video chunks of acceptable quality even when the estimated available bandwidth is constrained. The estimation is based on time spent on previous chunks transferred. This reactive estimate may mis-estimate available bandwidth because it is backward looking rather than informed.

Network latency is another major culprit. Applications rely on network protocols like TCP [41] and QUIC [78] to install a timeout and re-transmit when packets are delayed or lost. Tail latency is inevitable in unreliable network environments where packets delays and loss are common, because multiple timeouts are needed for a successful packet transmission, rather than swiftly reacting changes in network conditions. The protocol layer is oblivious to heterogeneity of the underlying network interfaces that may provide different latency distributions at different moments in time. To see why this is a problem, consider the typical cell phone connected via both WiFi and its cellular carrier. There are transient moments when WiFi will perform significantly worse than the cellular network, but the former is still the device's preferred network.

Locating elastic resources closer to applications can alleviate the aforementioned problems in some degree, because last-mile wireless links today offer high bandwidth [124, 125], low latency [124, 125, 54, 92], and good availability [107, 153]. However, the geographically distributed nature of resources in "last-hops" introduces new challenges for mobile applications. For stationary devices like smartphones and laptops, applications can continue to offload to servers in the same edge node. In contrast, experiments show that a mobile device using the "wrong" edge node can see worse performance than it would simply going to a back-end cloud service.

This is particularly important for emerging vehicular applications for semi-autonomous driving. Such applications tend to be latency-sensitive, and serving them from nearby edge nodes is often preferable to doing so from back-end cloud sites. Such applications can also be data-intensive. A recent article reports that one vehicle generates up to 25 GB sensor data per hour [69]. Processing such voluminous data at the edge is faster, imposes lower burden on backbone networks, and allows mobile devices to be more modest in capability. However, as those mobile devices move, the servicing node in the edge often needs to move with them; not doing so can be worse than relying on the back-end cloud in the first place.

As a whole, existing mobile systems are largely oblivious to turbulent network connectivity in three aspects. First, they cannot easily reason about diverse connection endpoints. Second, they do not have visibility into the network itself and its sources of turbulence. Third, they do not have adequate support for edge computing in the face of high mobility.

1.2 Thesis Statement

In light of these challenges, it is my thesis that:

Emerging distributed, connected application architectures suffer from turbulent network environments. Network-conscious systems for the applications are effective and beneficial.

To validate the statement, we first conduct a series of measurement studies to identify problems that applications encounter in today's turbulent network environments. Then, we present solutions to each of the problems in order to improve the application connectivity.

Chapter II describes the results of these measurement studies, focused on high-mobility vehicular settings. Section 2.1 measures the performance of current mobile wireless networks. In Section 2.2, we measure the performance improvement by using edge resources in vehicular settings. Section 2.4 summarizes the lessons from the measurement study and its implications.

Chapter III tackles the tail latency problem experienced by high-mobility clients. We propose *strategic redundancy* to improve interactive latency over such networks. Section 3.2 describes the design and implementation of our prototype, RAVEN. Section 3.4 discusses evaluation results for the prototype in both trace-driven emulation and live experiment setups. Section 3.6 lists out the main takeaways from RAVEN.

Chapter IV considers co-locating elastic resources closer to end-devices. We identify potential problems arising when using such *edge nodes* for high-mobility clients. In Section 4.1, we first explore various location prediction methods to support existing co-location systems. Section 4.2 shows that even the state-of-the-art system results in disappointingly bad application performance. As a potential remedy, Section 4.3 introduces a new system called Croesus that separates application-specific states to improve co-location time. Section 4.6 summarizes the problems in using distributed edge nodes for vehicles and the main contributions by Croesus.

Chapter V acknowledges that applications know best about adapting to connectivity turbulence, but the sources of that turbulence are often mid-network. We introduce a generic framework that allows effective application-aware adaptation in the middle

of such turbulent network paths. Section 5.2 details the design principles and choices we have made for our prototype framework, BumbleBee. Section 5.4 evaluates the prototype in three case studies with machine learning (ML), video streaming, and streaming-processing applications. Section 5.7 summarizes the main lessons learned in building BumbleBee.

Chapter VI concludes by outlining contributions this thesis makes as well as limitations encountered along the way. Finally, I enumerate potential research directions that can lead to future work.

CHAPTER II

Measurement Study

Application connectivity remains extremely turbulent despite a significant improvement in networking technologies. Increasingly, applications need more reliable and superb connectivity to seamlessly use elastic computing resources located across various points of network infrastructure. Although previous studies—wireless network performance [13, 32, 24, 130, 66], network performance in high-mobility environments [24, 85], performance of intermittent WiFi APs from high-mobility vehicles [20, 33, 109, 110, 58, 93], edge migration performance [54, 53, 92], and turbulent networking environments [141, 150, 50, 116, 14, 73]—partially answer some relevant questions about the implication of connectivity turbulence to application performance, in this chapter, we seek to extend the measurement studies to better understand expected application performance in today's networking environment.

We begin with a measurement study in a wireless networking environment, especially in a high-mobility setting. Because interactive applications are commonly used in the setting, we focus on measuring Round-Trip Time (RTT) to understand potential application-perceiving latency. The measurement results suggest that tail latency is very common. That leads to the next measurement study about reaching various points in network infrastructure. Cellular towers are so ubiquitous nowadays that we envision using cell-towers as intermediate edge nodes. We measure RTTs to reach resources in cellular towers and in cloud infrastructure. The measurement re-

sults clearly indicate that the latency to cloud infrastructure is extremely volatile. The latency to edge resources is promising only if co-located correctly.

We shift our focus from user-centric networking to a data-center environment, because many services run in and/or across data-center networks. Specifically, we measure bandwidth fluctuation in a commercial cloud-platform that over-provisions a number of tenants to generate more profit with a limited physical networking capability [28, 11]. The results suggest that even a data-center networking environment can be extremely turbulent. Finally, this chapter summarizes connectivity challenges that today's applications face. Then, it concludes by outlining main lessons learned that heavily govern the problems tackled and design principles, discussed in the following three chapters.

2.1 Measurement Study in a High-Mobility Networking

Vehicles sold today are equipped with multiple network wireless interfaces that provide distinct performance and availability patterns (e.g., WiFi, 4G, and Bluetooth). To simultaneously measure the latency over multiple network interfaces, we have developed a measurement tool, called VNperf. Once per second, the tool measures round-trip time (RTT) over multiple networks—three cellular and one WiFi hotspot—by sending a small RPC packet to a dedicated server located at the University of Michigan. The former interfaces provide large coverage; the latter provides intermediate coverage with low latency. The tool also collects the location and speed of the vehicle. The extended version of the tool, described later in Section 2.2, collects this additional information: the associated cellular tower identifier, association duration, and signal strength. We start with latency measurement in stationary settings with three cellular networks. We repeat the measurement in high-mobility settings with two cellular and one WiFi hotspot, by driving a vehicle in four representative driving scenarios: downtown, highway, rural, and suburb, around Ann Arbor, Michi-

gan. Each trace ranges 40-62 minutes long.

Recall that vehicles today have multiple network interfaces available. These heterogeneous interfaces may have distinct underlying performance and availability distributions. This section seeks to answer the following questions with the traces collected:

- How unpredictable is the latency over wireless networks?
- To what extent, does mobility contribute to volatile network performance?
- How challenging is it to predict the near-future network performance?

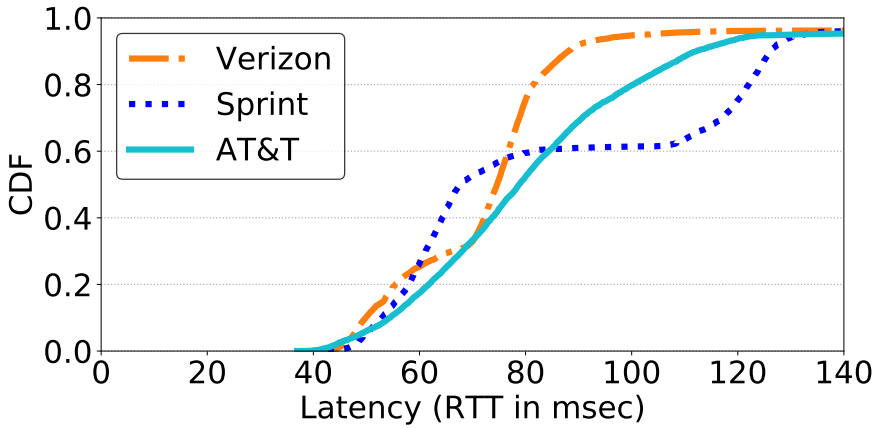


Figure 2.1: CDF of RTTs over three cellular networks for a stationary device.

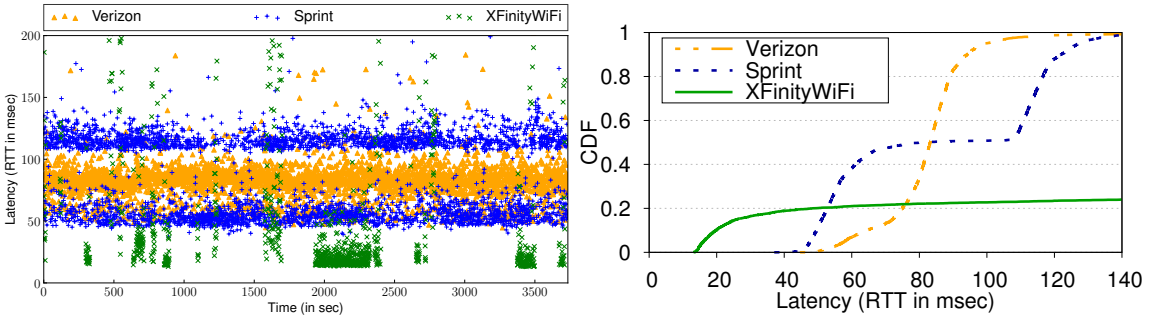
We first look the results of latency measurement in the stationary setting. Figure 2.1 reports the RTT distributions for three cellular networks in a stationary case. Immediately, we notice that the shape of each distribution is unique; Verizon resembles a normal distribution, Sprint has bimodal distribution, and AT&T is somewhere in the middle. Surprisingly, even though the device had approximately constant radio signals, the tails of distributions are high for all three interfaces. The 99% tails are 324 ms, 416 ms, and 389 ms for Verizon, Sprint, and AT&T, respectively.

We turn our attention to the high-mobility settings. Table 2.1 is a summary of RTT measurement results with additional information. Out of three networks, WiFi offers the lowest median latency when available. WiFi is available 25% and 30% in the downtown and suburban areas, respectively. Our WiFi results are a considerable change from a previous 2010 study [13] that reported results from an area correspond-

Trace	Mean MPH	Verizon			Sprint			XFinityWiFi Hotspot			
		Med.	95%	99%	Med.	95%	99%	Med.	95%	99%	Avail.
D1: Downtown	5.33	83	99	125	81	127	142	30	1166	4273	30.07%
D2: Highway	66.55	87	120	182	115	370	12569	N/A	N/A	N/A	0.00%
D3: Rural	35.71	85	174	2933	110	221	8533	N/A	N/A	N/A	0.00%
D4: Suburban	7.52	82	98	127	111	135	9019	48	516	2333	25.75%

Table 2.1: Median (Med.) and tail RTTs for traces collected in four driving scenarios. RTTs are given in milliseconds. The last column (Avail.) shows how often WiFi was available in each scenario.

ing to our downtown trace. That study found average vehicle-to-WiFi access through open APs to be available only 11% of the time. We found almost zero access through open APs, but commercial WiFi access was available 3 times more often than APs in the prior study.



(a) Scatter plot of downtown trace (b) CDF of downtown traces

Figure 2.2: Scatter plot and CDF of RTTs in downtown driving scenario. RTTs above 140 ms are not shown.

Interestingly, all three networks exhibit significant tail latency, ranging from 125 ms to above 12 s. Figure 2.2a is the scatter plot of the downtown driving trace. Clearly, no network offers a consistently superior performance. Furthermore, the latency of a network often varies substantially from second to second. This finding aligns with prior studies that report high variance in RTTs over cellular networks in high-mobility environments [24, 85]. On the other hand, the average RTTs of the networks are surprisingly stable. We confirm that there are very few trends where RTT averaged over a longer time window increases or decreases. In sum, the latency over a short time window is extremely unpredictable.

Figure 2.2b is the distribution of the same trace. Each network exhibits a distinct latency distribution. Verizon has an approximately normal distribution, Sprint has bimodal behavior, and WiFi has mixed periods of low RTTs and high latency. A recent study [104] finds that in a longitudinal measurement, the RTTs of the networks are stable and predictable, because the RTT distributions rarely change. In our traces, the high-mobility could have contributed to distinct shapes. So, we measure the impact of the mobility in RTT distributions by measuring RTTs in a stationary case.

We next examine the predictability of network latency. For each sample, we first determine which network offers the lowest RTT. We then ask: how often does the network with the lowest RTT in each sample offer the lowest RTT in the next sample (one second later)? The answer is: surprisingly infrequently. Across the four traces, the lowest-latency network at a given time remains the lowest-latency network one second later only 56% of the time. If the vehicle is stopped, predictability increases: 61% of the time, the lowest-latency network for a stopped vehicle remains the best network one second later. For vehicles in motion, we found no correlation between the vehicle's speed and how often the lowest-latency network would remain best one second later.

While many prior studies have examined wireless network performance [13, 32, 24, 130, 66], our specific focus is on examining network performance for vehicle-to-infrastructure communication and latency-sensitive applications. Thus, many prior studies measure bandwidth, while we measure round-trip time (RTT). Further, the findings in our study often differ significantly from those of prior vehicular studies. Balasubramanian et al. [13] study WiFi availability for offloading cellular traffic to WiFi. Deng et al. [32] and Chen et al. [24] study the goodput of MPTCP over WiFi and LTE. Sommers et al. [130] show high variability and tail latency for wireless networks.

Many prior studies investigate using intermittent open WiFi APs from fast-moving

vehicles [20, 33, 109, 110]. In contrast to these prior studies, we find that *commercial* WiFi offerings now provide reasonable coverage in densely populated areas and support moving vehicles with no modifications, yet *open* WiFi access points without splash screens or other authentication are very hard to find. Use of multiple networks for public bus WiFi has been studied [58, 93].

Our measurement study of high-mobility networking shows that wireless networks in high-mobility setups are fundamentally *turbulent*. Throughout different networks, the tail latency is common. Though the mobility contributes to a degree of turbulence, it is not the determining factor. Predicting the near-future latency is extremely challenging. Moreover, the best network that offers the lowest RTT remains only 3 s in median. The results in sum suggest that there is a missed potential improvement by using one network at a time.

2.2 Measurement Study to Various Points of Infrastructure

We next measure RTTs to reach various computing points located across network infrastructure. Specifically, we measure RTTs to three points: an associated cellular tower, a *nearby* but not associated tower, and cloud. We measure the inter-tower latency by running VNperf on two mobile devices. Also, we collect the handover traces in high-mobility settings by collecting tower association history in three representative scenarios in Ann Arbor, Michigan: downtown, highway, and rural.

The results in the previous section suggests that the latency reaching back to the cloud can be unacceptable for highly interactive applications such as AR/VR applications. Because the recent advances in wireless technology have enabled low stable latency over the last-mile links, the edge-computing resources, located at the last-mile hop, can be a potential solution for those types of applications. The promise of edge computing relies in part on co-locating computation close to where it is needed. In this section, we examine the current cellular deployment to answer several

important questions:

- How much is latency overhead imposed when the "wrong" edge nodes are used?
- How often does an edge computation need to migrate?
- How constrained is the inter-tower bandwidth?

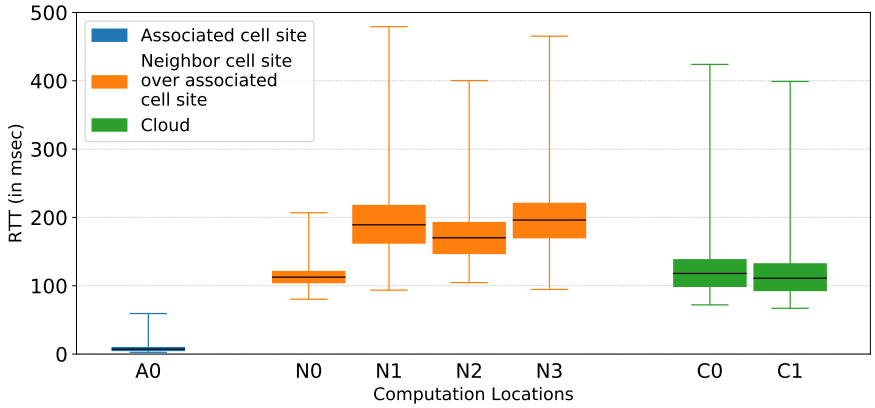


Figure 2.3: RTTs from mobile client to Associated cell site, to nearby Neighbor cell sites, and to back-end Cloud sites. Each box-plot represents the 25th and 75th percentiles with black median bars and minimum and maximum wisps.

Figure 2.3 shows the latency experienced by a mobile client when communicating with edge computations at three potential edge sites in stationary settings. The median RTT to the associated tower is only 7.2 ms. In contrast, the median RTT to the *nearby* yet not associated towers ranges from 112.5 ms to 196.2 ms. Surprisingly, the median RTT to the cloud is only 78.8-118.0 ms. In other words, for latency-sensitive interactive applications, co-locating edge computations at the "wrong" cellular tower can be much worsen than relying on computations at the cloud.

We further investigate on the disappointingly high median RTT to neighbor towers located a few miles apart. In our measurement, one of neighbor towers (N0) is located only a mile apart from one that the mobile client is associated with. From our packet logs, we discover that data packets traverse fairly deeply into the backbone network before they are rerouted back to the neighbor tower. The route between the two sites includes multiple network hops such as a metro Ethernet network, a regional telephone switching office, and a national switching center. As a result, the packets

transverse at worst 1,600 miles end-to-end, even if source and sink towers are only a mile apart.

Trace	Duration	Mean MPH	Number of Towers Association Changes	Association Duration		
				Median	95%	99%
D1: Downtown	42 mins	13.80	65	18s	84s	171s
D2: Highway	44 mins	60.90	69	25s	158s	465s
D3: Rural small town	48 mins	29.86	41	27s	325s	511s

Table 2.2: Summary of number of tower association changes and association duration per tower over three representative driving scenarios.

If using "wrong" edge nodes can be painful for highly interactive applications, how often do we need to move the computations? To answer that, we measure association duration in the high-mobility settings. Table 2.2 is the summary of tower association changes. We immediately notice that the number of changes in tower association in all three cases are high. However, the tails of association durations are quite different, ranging from 171 s to 511 s. In other words, an edge node can be used anywhere from as low as several seconds to as high as nine minutes.

Day	Tower Association Sequence
1	9 1 2 7 1 8 3 5 3
2	1 2 1 2 1 3 4 3
3	1 2 1 3 4 3 5 3 5 3
4	1 2 1 2 1 2 1 3 6 3 4 5 4 5 3 5 3
5	1 2 1 2 1 3 4 3 4 5 3 5 3 5

Table 2.3: The sequence of tower associations in driving a short commuter route repeatedly at five different dates over three months. Traversing the route (approximately 6 miles) takes 10 minutes in a good traffic.

We further examine patterns in association changes. On the same route, if the sequences of association changes are similar or identical, moving computations ahead-of-time is easy. To confirm our hypothesis, we collected the handoff traces by repeatedly driving the same 5.5 mile route on five different days spread out over several months. Table 2.3 shows the sequences of tower associations for each of these drives. When traveling this same route repeatedly, we see that while the tower sequences follow sim-

ilar broad patterns (e.g., `1` appears before `3` appears), the specific sequence changes each time. In some cases, an individual tower appears on only one of the five days. There is likely to be some predictability for a given path, but it will not be perfect.

Inter-tower bandwidth capacity is critical for moving edge computations when a mobile client is being handed off. Compared to the data-center network, the network bandwidth in cellular edge sites can be extremely constrained. In fact, inter-tower bandwidth capacities are often held over from previous generations of cellular network. Also, the topology of towers is hierarchical. So, the capacities from towers to back-haul network can be a potential bottleneck. We analyze raw capacity data for more than 117,000 towers provisioned nationally and operated by a major cellular provider. The median capacity of inter-tower bandwidth is only 430 Mbps. Worse, 10% of sites have capacity less than half of the median.

Such highly constrained bandwidths can lead to high migration delays when moving edge computations. For example, two interactive vision applications – Eye tracker and AR navigation – have image sizes of 1 GB and 600 MB to transfer, respectively. Therefore, just moving the full image from one site to another across median-capacity links takes on the order of 10 seconds. Moving only the dynamic portion of the image by computing the delta and shim of VM [54] and docker [92] images is possible; in our experience with edge-native applications, 10 seconds of active computations in service generates dirty state on the order of 10-100 MB. Hence, even with such optimizations, it still requires on the order of seconds to fully transfer images between cellular sites.

In summary, emerging vehicular applications can greatly benefit from using resources in cellular edge nodes, compared to ones in a data-center. The distributed nature of edge nodes is challenging, because the associated tower changes so frequently and unexpectedly. Latency to "wrong" nodes, i.e., to a previously associated node, is higher than to the cloud. To retain low latency, a computation server must migrate to a newly associated tower. Unfortunately, our study shows that inter-tower

bandwidth is prohibitively constrained. Migrating state between cellular edge nodes may take an order of seconds, resulting in worsened application performance and poor user-experience.

2.3 Measurement Study in Data-Center Networking

Over the turbulent wireless networks, applications reach the cloud infrastructure. Unfortunately, turbulence remains perturbing across accessing distributed computing resources in the cloud infrastructure [50, 116, 14, 73]. This is especially troublesome for bandwidth-heavy applications that desire stable, high bandwidth connectivity for many distributed computing nodes, e.g., stream-processing jobs and video-streaming servers. As noted by previous works [141, 150], temporal bandwidth dip can hurt application performance and user-experience drastically.

We have measured the available bandwidth between two virtual machines (VMs) located in the same data center. This is probably the most common setup that distributed back-end applications are deployed. However, in some cases, computing nodes are located across different data centers for various reasons such as cost to move data and the General Data Protection Regulation (GDPR). So, we have repeated the measurement study with two VMs, one located in East and another located in West, to simulate the inter-connectivity of computing nodes between data centers. For the measurement, we have used `iperf3` tool that reports measured throughput at every 5 s interval.

Figure 2.4 shows the measured bandwidth of two cases over an hour window. We immediately notice is that inter-connectivity across data centers (Inter-DC) has high variability in a short time window. Applications may experience transfer time that varies by 42% when moving an identical-sized object. This can be problematic for geo-distributed applications that transfer a large data across data-centers. The next observation is that intra-connectivity within the same data-center offers mostly stable

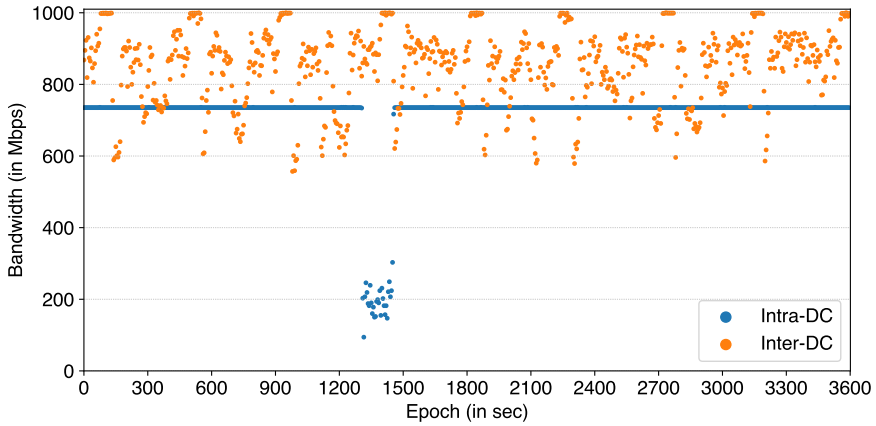


Figure 2.4: CDF of RTTs over three cellular networks for stationary device.

bandwidth, 735 Mbps in median. This is expected because two VMs can be hosted in the same lack or even in the same physical machine [144]. Yet, around epoch 1200-1500 s, we observe a large bandwidth dip, and bandwidth stays between 92-200 Mbps. Abrupt bandwidth dips can be troublesome for distributed connected applications such as a live video-streaming service that needs to meet a tight processing time budget.

In sum, the measurement results show that both inter- and intra-connectivity in data-center networking environments have a degree of turbulence in bandwidth for different probable reasons. The results also suggest that building a network-conscious system for a data-center environment can be effective for distributed, connected applications to adapt.

2.4 Lessons

Our measurement studies clearly show that application connectivity remains highly turbulent in both mobile and data-center environments. In particular, high-mobility settings come with latency problem. User-facing applications typically used in vehicles need a superb connectivity to infrastructure such as cloud- and edge-computing. Vehicles sold today are equipped with multiple network interfaces. So, applications can choose to send over multiple network interfaces. However, the study

results show that no network consistently offers superior RTTs. Instead, each network has a distinct RTT distribution and availability. Tails of RTT distributions for each network are extremely high. As a result, predicting which network will offer the best performance in the near-future is quite challenging. The findings suggest that the latency from vehicle to infrastructure can result in high tail latency. Fortunately, the study also suggests that one network with high RTTs can be masked by using another network, when redundancy is employed.

Wireless networks are fundamentally turbulent. One potential solution is moving elastic computing-resources closer to end-devices. We envision using ubiquitous cellular towers as intermediate locations for edge-computing resources. The study results show that when end-devices and edge-nodes are co-located correctly, i.e., computation server is at an associated tower's edge node, the latency benefit is clear, offering an order of magnitude lower RTT than to cloud. However, the distributed nature of edge nodes in conjunction to vehicle's high-mobility makes it challenging to co-locate two parties correctly. Migrating stateful server may result in prohibitively high downtime due to constrained inter-tower bandwidth provisioned. Hence, the study results motivate us to investigate migration strategies and technologies in the context of cellular edge nodes for vehicular applications.

Edge-nodes are intermediate complementary solutions; connected applications ultimately need to interact with the cloud infrastructure to stream new feeds, to fetch and publish new data, and to connect with other entities. Sadly, the data-center network environment also remains highly turbulent. This can be problematic emerging distributed applications that run across multiple physical machines in a data-center or across multiple data-centers. Our study results align with previous findings that the data-center network environment is highly turbulent despite well-provisioned bandwidth links. As a result, the end-to-end application connectivity may experience high latency in and from the data-center network environment, leading to worsen user-

experience.

In conclusion, we have identified connectivity problems that applications encounter today. The overall study results motivate us to build *network-conscious* systems to improve application performance over turbulent network environments. In the following sections, we present three systems to conquer the problems and to improve application connectivity, turn by turn.

CHAPTER III

Redundant Network Transmission

The latency measurement study clearly shows that predicting near-future RTTs of individual wireless networks is extremely challenging. A wrong prediction of the best network may hurt the performance of latency-sensitive interactive applications, and result in poor user-experience. Fortunately, modern end-devices are equipped with multiple network interfaces, e.g., cellular, WiFi, and Bluetooth interfaces. Often, these interfaces exhibit distinct performance and availability distributions. In other words, one network sometimes performs better than others but not always. Accurately predicting which interface offers the lowest RTT in the near-future is difficult; redundantly transmitting data over multiple heterogeneous interfaces, and using whichever arrives the fastest, the application-perceiving latency can be improved. Therefore, in-network redundant transmission can be *beneficial* to latency-sensitive applications.

However, always transmitting redundantly can be wasteful, especially when one network constantly outperforms others. So, a scheduler must employ redundancy strategically to minimize excessive resource consumption. This chapter describes the core design principles to employ in-network redundancy strategically only when it is necessarily. It starts with a discussion of an existing multipath system. Specifically, it focuses on a state-of-the-art system, Multipath TCP (MPTCP). Then, it elaborates the problems centered around employing in-network redundancy strategically in exist-

ing systems. Next, it details a prototype scheduler called RAVEN that incorporates key design principles. Finally, it illustrates the performance improvement made by RAVEN in both trace-driven simulation and live experiments. The chapter concludes with some questions that have not been answered.

3.1 Background: Multipath TCP

Multipath TCP (MPTCP) is the state-of-the-art system that multiplexes a single socket connection over multiple low-level TCP *subflows* [57, 122] that traverse different network routes. Each subflow corresponds to a different wireless network interface. The default scheduler in MPTCP is a `minRTT` that sends each packet over one network at a time. For each packet transmission, the scheduler selects the subflow with the lowest predicted RTT among all subflows that have not yet reached their congestion window. Once the minimum RTT network reaches the congestion window, the next-lowest RTT network is used. So, small size data are typically sent over the network with the lowest predicted RTT; Large transmissions are *striped* over multiple networks.

MPTCP links are bidirectional; both endpoints have independent schedulers. MPTCP is flexible; new schedulers can be implemented as a loadable kernel module. This has helped to make MPTCP scheduling an active area of innovation for industry [3, 77] and the research community [56, 88, 87] (e.g., the latest release of iOS has 3 separate MPTCP scheduling policies [3]).

3.2 Design Principles for Strategic In-network Redundancy

In-network redundant transmission has a potential latency improvement: it sends data over multiple networks redundantly, uses whichever that arrives the earliest, and discards later arriving ones. However, this approach may incur excessive data usage when employed constantly. To systematically hedge a bet against uncertainty,

strategically employing redundancy only when it is needed is *beneficial*. This section investigates three core problems that heavily influence the design principles to support the *in-network* redundancy.

3.2.1 Problem: Stale RTT measurements

MPTCP calculates an RTT estimate for each TCP subflow. It collects samples from passive measurement of previous transmissions. It computes an exponentially weighted moving average (EMWA) over collected samples to estimate current RTT. Eventually, the estimate weights samples based on the *order* they were collected [70]. For applications that transmit large data frequently, this is a reasonable choice because there are many samples collected in a short time window.

Unfortunately, interactive applications transmit small amounts of data infrequently for various reasons (e.g., waiting on user input, utterance streaming). As a result, the wall-time epoch of collected samples can be far apart, even though the order is relatively close. In other words, an RTT sample measured an hour ago may have equivalent weight as sample measured a few seconds ago. And the weight of stale sample can result in inaccurate RTT estimation. Therefore, weighting RTT samples based on wall-time epoch adjusts weights of stale measurements.

3.2.2 Problem: Scalar RTT prediction

Existing schedulers rely on *scalar* RTT prediction to select which network to use. However, in a multi-network environment where a network remains the best network for only 3 s in median, a network's performance fluctuates a lot in a short time window. When multiple networks are available, the scalar prediction does not account for the *uncertainty* of the near-future RTT variants. So, existing schedulers often mis-predict the near-future best network, and this results in worsen latency.

Although wireless networks have very short run-lengths, the underlying a network's RTT distribution remains relatively stable over the long-term. However, we

cannot assume a particular type of distributions when there are sufficient measurements for the central limit theorem to hold. Instead, using the distribution of past relative prediction errors, we can estimate certainty in the form of a *confidence interval*. A confidence interval quantifies the certainty of current RTT predictions.

3.2.3 Problem: Active probing is wasteful

A confidence interval is insufficient to overcome infrequent samples for interactive applications. This is because a previous network that performs the best will be selected even though another network's performance has improved. A potential approach to conquer this is periodic active probing that sends small data over unused networks at fixed intervals. However, this can be very wasteful when a network continuously underperforms.

Instead, we propose an *aging* mechanism that extends the tails of the confidence interval, proportional to the last-used time. Aging is a more elegant solution than periodic probing; it employs redundancy over an unused network to collect recent samples and to hedge a bet on the potential RTT improvement of the network.

3.3 Design and Implementation of RAVEN

We have implemented a prototype scheduler, called RAVEN[1] that incorporates the core design principles discussed in the previous section. This section outlines some of the design choices made for implementing RAVEN.

3.3.1 Adjusting for stale RTT measurements

TCP weights RTT samples by the order in which they arrive; e.g., the nth sample receives the same weight if it has taken 100 ms or 10 seconds in the past. To support interactive applications that transmit small amounts of data infrequently, RAVEN uses elapsed time rather than order to weight RTT samples. RAVEN uses a weighted

[1]Redundancy Aided VEhicular Networking

average of per-subflow RTT samples to predict RTT for each subflow:

$$RTT_{pred} = \frac{\sum_i^n w_i * RTT_i}{\sum_i^n w_i} \tag{3.1}$$

RAVEN weights the ith RTT sample by the time elapsed since the sample was taken:

$$w_i = e^{-(t_{now}-t_i)/\lambda} \tag{3.2}$$

where t_{now} is the current timestamp, t_i is the time the RTT sample was taken, and λ is a network-specific aging factor. Thus, two consecutive samples will have almost the same weight if they occur within a short time period and very different weights if they are separated by a long time.

RAVEN adaptively sets λ to minimize the root mean squared error (RMSE) of the prediction error for previous predictions made for each network. It temporarily logs each prediction and the actual RTT measured by TCP. We observe that λ changes infrequently, so RAVEN recalculates values once per day based on the previous day's observations.

3.3.2 Calculating confidence interval

RAVEN always sends data over the network with the lowest predicted RTT, similar to the default MPTCP scheduler, `minRTT`. However, When predicting RTT for each subflow, RAVEN estimates certainty in the form of a confidence interval calculated using the distribution of past relative prediction errors. If the RTT confidence interval for any other network overlaps with the confidence interval of the lowest RTT network, RAVEN also transmits the data over that network. This yields strategic redundancy: RAVEN uses multiple networks when it has poor confidence that one will be faster than another.

RAVEN calculates confidence intervals from the order statistics of relative prediction errors. For each sample, it computes relative error by dividing the actual RTT measurement by the value it predicted. We tried using both relative error and absolute error and found that relative error yielded slightly better results. Specifically,

RAVEN sorts the relative errors and takes, e.g., the 5% and 95% percentile values to calculate the 90% confidence interval over the relative error. Confidence intervals are recalculated once per day when RAVEN recalculates λ.

Confidence intervals provide an effective knob for adaptively tuning the trade-off between performance and data usage. An application or user can choose which confidence interval RAVEN should use and thereby adjust the trade-off between minimizing RTT and minimizing data usage. Intervals of 100% cause data to be sent over all available networks; intervals of 0% yield no redundancy. RAVEN uses 90% confidence intervals by default.

3.3.3 Replacing active probing with aging

RAVEN uses its confidence in its predictions to determine when to transmit data redundantly over a network. Intuitively, if our last measurement of a poor-performing network is recent, we have high confidence in that prediction. As time passes without new measurements, our confidence in the prediction decreases. Eventually, the confidence interval for the poor-performing network will overlap with that of the best-predicted network, and RAVEN will transmit data redundantly over the poor-performing network, generating new RTT measurements. This provides a more strategic replacement for active probing.

RAVEN scales each confidence interval by a factor that is a function of the time since the most recent sample was taken. Note that the recency of samples does not affect the expected value of predictions since the exponential weightings of all prior samples decrease by the same relative amount as time passes. The scaling function is calculated from empirical observations once per day. RAVEN first calculates the RMSE for past samples. To calculate the scaling factor for the case where the most recent sample is n seconds old, it first calculates a prediction for each sample omitting any measurements that occurred less than n seconds prior to the prediction. It then calculates the RMSE for those predictions. The relative difference between RMSEs

is the scaling factor for time n. RAVEN stores the scaling factor for different values of n in a lookup table. In summary, the bounds of the confidence interval for a RTT prediction are:

$$RTT_{pred} \pm (RTT_{pred}/RelError_{CI}) * Age(T_{now} - T_n) \tag{3.3}$$

where $RelError_{CI}$ is the confidence interval of the relative error calculated as described in the previous section, $Age()$ is the scaling function, T_{now} is the current time, and T_n is the most recent RTT measurement. Note that $RelError_{CI}$ and $Age()$ are calculated once per day, but the other variables are based on recent RTT measurements.

In contrast to periodic probing, RAVEN will not try to use poorly-performing networks when it is confident that a current network offers better performance. For example, with a stable low-latency WiFi connection, it makes little sense to see if a poorly-performing cellular network has improved; even the best case latency for the cellular network will not be superior to that of the currently-available WiFi.

3.3.4 Identifying latency-sensitive traffic

Rather than require application hints, RAVEN automatically detects traffic for which low RTT is relatively unimportant and avoids redundancy in those instances. Small transfers are most sensitive to RTT. Large transfers (e.g., downloading a binary or video) are potentially latency-sensitive, but the impact of RTT on the relative completion time of the entire transfer is insignificant; the default MPTCP scheduling policy of striping across multiple networks is likely best [121, 122].

For each MPTCP connection, RAVEN automatically switches between redundant mode (in which it may transmit the same data over multiple networks) and non-redundant mode (which, in order to limit implementation complexity, is the same as the default MPTCP scheduler). Since RAVEN is implemented as a kernel module, it can observe several low-level data structures to attempt to differentiate between small and large transfers. We examined several options and determined that the per-subflow

TCP queue size, per-flow congestion windows, and main MPTCP socket queue occupancy were the most informative metrics to consider. The RAVEN scheduler switches from redundant to non-redundant mode if at least $n-1$ out of n active subflows have their congestion window's worth of data in the per-subflow queue. It switches from non-redundant to redundant mode if all per-subflow queues and the main MPTCP socket queue have less than two packets to transmit.

With this algorithm, distinct transfers usually start in redundant mode. Small transfers typically complete before a mode switch happens, and large transfers switch after sending several packets. Our results in Section 3.4.5 show that this strategy results in very little data being sent redundantly for typical large consumers of data such as Web, video, and application downloads. Since large transfers naturally contribute the vast majority of bytes sent, most data sent by RAVEN will be transmitted non-redundantly. Yet, RAVEN still improves response time significantly for interactive applications with small transmission sizes.

We have also built an MPTCP proxy so that applications that are not MPTCP-aware can use RAVEN without modification. A client proxy, which we expect to run on the connected car's AP, accepts TCP connections and redirects traffic to either MPTCP-aware servers or a persistent MPTCP connection with a cloud proxy. The cloud proxy, in turn, redirects traffic to unmodified servers.

3.3.5 Cancelling duplicate transmissions

Prior user-level implementations [51, 59] have limited ability to cancel useless work (e.g., sending data over one network that has already been acknowledged by another) because they cannot easily revoke data that has already been given to the kernel. This leads to head-of-line blocking, and may require throttling to improve interactive latency [60].

MPTCP requires each subflow to deliver data in order; it pushes data from the subflow queue to its meta-level queue only when the data is in order according to the

subflow sequence number. In contrast, RAVEN also pushes data to the meta-level queue if data that is out of order at the subflow level would be in order at the meta level, and this avoids unnecessary retransmissions. Note that applications still receive data in order.

When sending acknowledgments, RAVEN inspects the MPTCP socket to see if missing packets have been delivered via other subflows; it includes such data when calculating the acknowledged sequence number, which in turn eliminates unneeded subflow-level retransmission. Finally, at the sender, when data is acknowledged by *any* subflow, RAVEN removes it from all per-subflow queues. If the data has not yet been sent by a subflow, this completely eliminates the redundant transmission. Otherwise, this avoids possible retransmissions.

Proactive cancellation is especially useful when RTT spikes are frequent or networks are intermittently available. For instance, when a network is temporarily unavailable, many packets can accumulate in the subflow queue. Eventually, this data is sent and acknowledged over another subflow. RAVEN proactively removes these packets when acknowledgments are received, so the subflow can transmit new data when connectivity returns. A user-level implementation cannot cancel this work, so considerable time and bandwidth are wasted transmitting useless data.

3.3.6 Implementation

RAVEN scheduler is implemented as a loadable kernel module. RAVEN works with the *mainstream* Linux kernel as of 2020. The kernel module implements the MPTCP `next_segment()` and `get_subflow()` functions to enforce its scheduling decisions and direct packets to the appropriate subflow queue. It maintains a shadow data structure, the *redundant queue*, that records which subflows are responsible for transmitting which data, along with the per-subflow transmission status. Currently, RAVEN clones packet headers for data in the redundant queue; this avoids conflicting with packets counting for TCP Segment Offloading (TSO). RAVEN uses integer

approximation of floating point calculations. For the exponential function, it uses Schraudolph's approximation [127]. Confidence intervals derived from order statistics and aging factors are stored in lookup tables to improve performance.

3.4 Evaluation

Our evaluation answers the following questions:

- How much does RAVEN improve performance for interactive applications compared to default MPTCP?
- Are confidence intervals an effective method for balancing performance and data usage?
- How effectively does RAVEN conserve bandwidth by avoiding redundancy for larger transmissions?
- How much do individual elements of RAVEN's design contribute to its performance improvements?

3.4.1 Methodology and Applications

We use both trace-driven emulations and side-by-side comparisons in live vehicle experiments. Section 3.4.3 reports live vehicular experiments; for repeatability, all other experiments use trace-driven emulation of the four network traces from the study in Section 2.1. We use a leave-one-out method in which we first train RAVEN to determine λ, confidence intervals, and scaling factors (discussed in Section 3.2) for each network using three traces, and then we evaluate the results of running RAVEN on the remaining network trace. Note that these traces were selected to be dissimilar, so this methodology biases against RAVEN to some degree.

Our emulated setup has a client with multiple interfaces and a server that responds to client requests. All machines run Ubuntu 14.04. The client has 8 3.5 GHz cores and 16 GB RAM; the server has 8 2.8 GHz cores and 8 GB RAM. We use TC to

replay collected traces, changing the RTT for each network every second to match the RTTs measured in Section 2.1. Note that since we measured RTTs every second, our emulation results do not include delays from network power management that occur after multiple seconds of network idleness. We evaluate the impact of power management in live vehicle experiments in Section 3.4.3. Since we are evaluating latency-sensitive applications that send only a few bytes, bandwidth values do not particularly affect our results. We set bandwidth to 2 Mb/s.

For live vehicular experiments, we run the same application simultaneously on two identical laptops, configured as described above. Although these experiments are inherently unrepeatable, we try to keep characteristics similar to a specific trace by driving in the same geographic area in which we collected the trace. We also synchronize the two applications so that they initiate requests at the same time.

We use three applications to evaluate RAVEN: speech recognition, music streaming, and Yelp recommendation. We selected these applications because they are commonly used in a vehicle; they also have a diversity of network access patterns. Each application records the response time of its activities. Unless otherwise noted, the applications are not modified to use RAVEN or provide hints.

3.4.2 Application response time

We begin by evaluating how much RAVEN improves application response time compared to the default MPTCP scheduler. Figure 3.1 shows results for speech, music, and Yelp, from left to right. Results for the four traces from our study in Section 2.1 are shown from top to bottom.

RAVEN uses its default confidence interval of 90%; in Section 3.4.4, we explore the effect of changing this parameter. Since our applications do not send much data, MPTCP does not stripe data and instead sends all data over the subflow with the lowest predicted RTT. Thus, its behavior should be the same as other schedulers such as Apple's modified minRTT scheduler [4]. We also show performance using TCP over

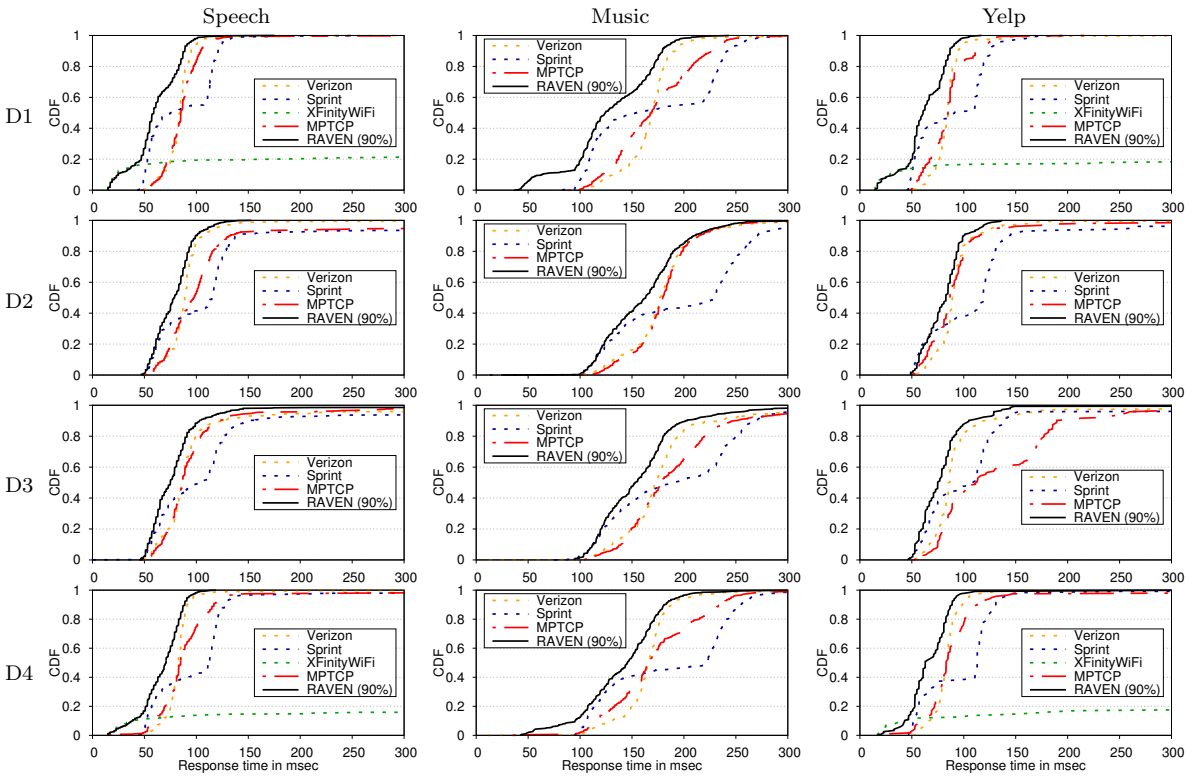


Figure 3.1: CDFs of application response time for speech, music and Yelp. We compare RAVEN (using its default 90% confidence interval) with the MPTCP default scheduler, as well as with TCP over cellular and WiFi.

each network.

RAVEN substantially outperforms default MPTCP for all applications in all scenarios, as shown by the CDF line in each graph being further to the left. For example, in D1:Speech in the top left corner of Figure 3.1, RAVEN is able to effectively use WiFi when it offers low latency, but MPTCP often does not use the network due to stale, high RTT measurements. At higher latencies, RAVEN has lower response times than either cellular network because it exploits whichever offers better connectivity at the moment. Across all scenarios, RAVEN provides an average speedup in a median response time of 26%, 19% and 30% for speech, music, and Yelp, respectively, as compared to MPTCP.

We note that these results seem to track how frequently each application transmits: the music application's network usage is most continuous, whereas Yelp's is the most

App	Trace	Verizon			Sprint			MPTCP			RAVEN		
		Med.	95%	99%	Med.	95%	99%	Med.	95%	99%	Med.	95%	99%
Speech	D1	84	101	119	75	125	141	85	112	132	59	93	109
	D2	89	119	173	115	2240	T/O	98	414	T/O	79	111	129
	D3	87	215	T/O	105	820	T/O	87	156	781	75	127	T/O
	D4	83	97	155	113	135	T/O	85	129	T/O	71	93	103
Music	D1	169	201	242	154	255	283	168	236	275	124	188	213
	D2	177	240	341	228	300	1056	180	232	281	163	230	260
	D3	173	289	825	190	296	657	179	315	867	153	244	317
	D4	167	205	312	220	265	436	167	246	339	146	195	245
Yelp	D1	85	101	185	91	131	169	85	125	137	61	93	105
	D2	89	129	175	119	265	10677	87	145	579	83	117	135
	D3	86	161	815	109	149	5565	112	257	713	75	129	149
	D4	83	105	135	113	141	251	83	131	558	63	93	111

Table 3.1: Median, 95%, and 99% application response times for speech, music, and Yelp. All values are in msecs.

infrequent. Our belief is that RAVEN's explicit incorporation into its predictions of when each measurement was taken yields greater improvement for applications with less frequent network usage.

Table 3.1 provides more detail about tail behavior for the individual cellular networks, MPTCP, and RAVEN (we omit WiFi since it is not frequently available for any entire trace). These results show that both RAVEN and MPTCP substantially improve tail response time compared to the individual networks by using a different network when performance on a given network is poor. However, RAVEN also substantially improves on MPTCP response times. Across the four scenarios, RAVEN provides an average speedup in 95% tail response time of 66%, 20% and 30% for speech, music, and Yelp, respectively, as compared to MPTCP.

For 99% tail response time, the speech application times out for over 1% of the recognitions in two scenarios for MPTCP, but only one for RAVEN. RAVEN provides a speedup of 21% over MPTCP in the other speech scenarios. Across all scenarios, RAVEN provides an average speedup in 99% tail response time of 52% for music and 341% for Yelp as compared to MPTCP. Thus, RAVEN reduces the substantial delays at the tail of the distribution.

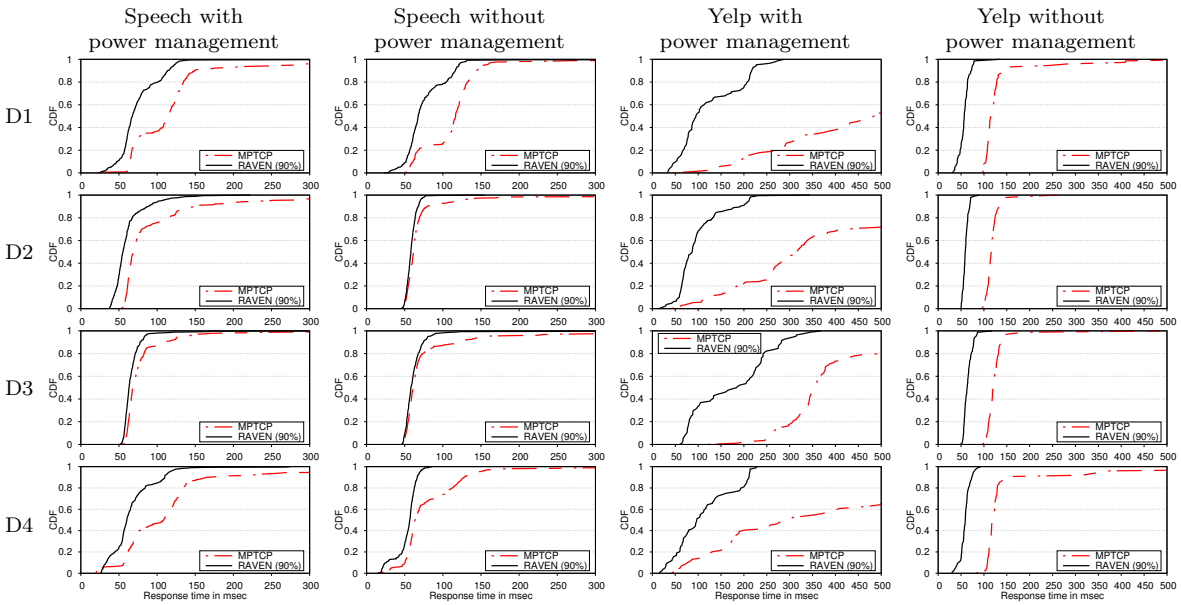


Figure 3.2: CDFs of application response time for speech and Yelp in live experiments.

3.4.3 Live experiments

We confirmed our emulated results by repeating a subset of the experiments in the vehicle. With two laptops available, we chose to compare RAVEN running with default 90% confidence intervals and MPTCP using its default scheduler. Figure 3.2 shows results for speech and Yelp in live scenarios of about 1 hour duration in a subset of our four traces.

Interestingly, RAVEN's relative performance benefit in the live experiments is greater than in the emulated experiments. More variation occurs in a vehicle; e.g., disconnected networks and poorly performing networks are not always distinguished correctly, and the frequency of RTT variation is greater than the one-second variation in our traces. RAVEN is designed to deal with uncertainty, and so it adapts better to this variation. For speech, RAVEN's average speedup in median response time over MPTCP is 46%, ranging between 8% and 80% for the four scenarios. RAVEN's average 95% response time is over two times faster, and its average 99% response time is over three times faster. MPTCP performs extremely poorly for Yelp in both scenarios. RAVEN's average speedup in median response time over MPTCP is 3.4x.

There is over an order of magnitude difference in both 95% and 99% response times.

From packet logs, we found the reason for this behavior: the default MPTCP scheduler interacts extremely poorly with network power management. Yelp transmits infrequently, so when a network is selected, it is often in power-saving mode. When sending a request to the server, both RAVEN and MPTCP experience a power promotion delay, which is the time for the network to resume sending data when it was previously in a low-power mode. However, MPTCP often incurs a second power promotion delay because the server chooses to send the response over a different network. RAVEN often avoids this cost because it sends the original request over two or more networks; the second network has already exited from its low-power mode when the response is transmitted. For speech, MPTCP will often incur a power promotion delay when switching to a new network; RAVEN often hides this delay by transmitting redundantly during periods where the lowest-latency network is changing.

A second poor interaction is that MPTCP's retransmission mechanism is unaware of power management delays; as a result, it can be either too aggressive or too conservative. The default MPTCP scheduler retransmits data over additional subflows if the subflow-level retransmission timer expires. The timer is conservatively set by multiplying the RTT estimation by 4 (see [123] for details). Thus, when the default MPTCP scheduler selects the wrong subflow, it can wait a long time before retransmitting over other subflows (i.e., it is too conservative). However, when a power promotion delay occurs, the default MPTCP scheduler will observe a long delay in receiving an acknowledgment and incorrectly switch to another subflow. In turn, that subflow will likely experience a power promotion delay. Because the default scheduler is not aware of power management delays, it is too aggressive in switching between subflows. Strategies that aggressively retransmit tail data [38, 155, 1] would help with the first problem, but also would likely make the second problem worse unless they explicitly consider power management behavior.

We confirmed these observations by re-running all the experiments while injecting a small amount of additional traffic to keep all networks from entering power save mode. The second and last columns of Figure 3.2 show results for speech and Yelp, respectively. As expected, the speech results are roughly similar (especially given the lack of repeatability in live driving), but the Yelp results show considerable improvement for MPTCP, and, to a lesser extent, for RAVEN. Still, for Yelp, RAVEN provides median application response time more than two times faster than MPTCP, while 95% and 99% tail response times are approximately four times faster.

3.4.4 Effect of changing confidence intervals

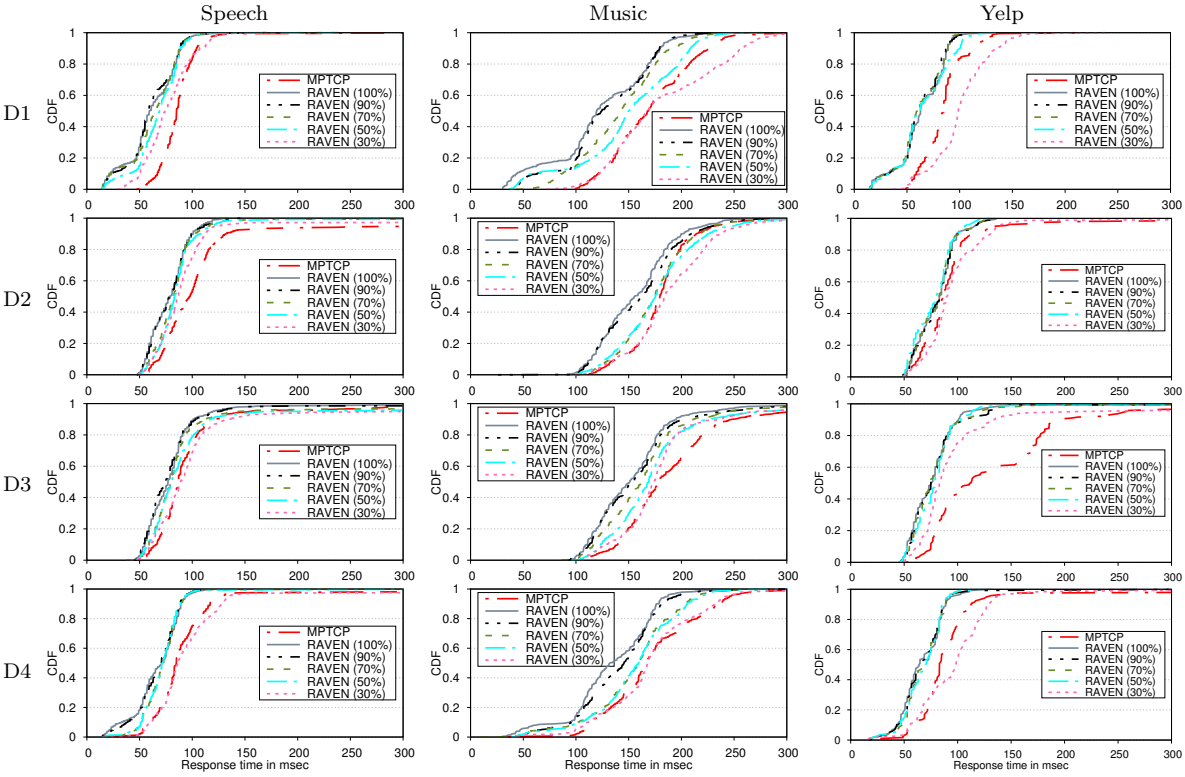


Figure 3.3: These CDFs show how the choice of confidence interval affects application response time for speech, music and Yelp. We show results with the default MPTCP scheduler for comparison.

Figure 3.3 shows the effect on RTT of modifying which confidence intervals RAVEN uses. Figure 3.4 shows the extra bytes sent over the network (normalized

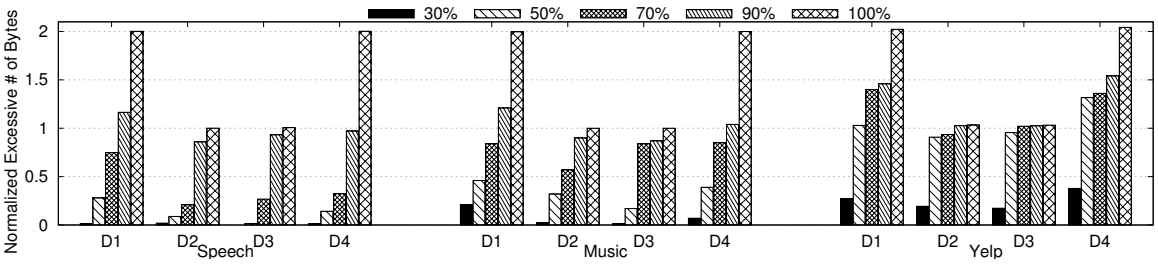


Figure 3.4: This graph shows how the choice of confidence interval affects extra bytes sent by the speech, music and Yelp applications. A value of 0 indicates no data was transmitted redundantly, 1 indicates twice as many bytes were sent, etc.

to the total bytes transmitted by the application) for each confidence interval. These and subsequent experiments are run in the emulated environment.

It is important to note that these experiments are designed to generate only latency-sensitive traffic with small data sizes. Thus, these values do not reflect how much redundant data would be sent for a typical workload. As we will see in the next section, RAVEN transmits almost no data redundantly for larger transmissions, and, due to their size, one would expect such transmissions to comprise the vast majority of bytes sent and received by a mobile device.

100% confidence intervals are always redundant: all bytes are sent over all networks. This offers the lowest RTTs, but it also effectively doubles the number of bytes sent when two networks are available and triples it with three networks (actually, the total number of bytes is slightly larger due to retransmission). Shrinking the confidence intervals results in successively fewer bytes sent, but also higher RTTs. While users may have different preferences, we chose 90% confidence intervals as the default setting for RAVEN because it lies at the knee of the curve. This setting has similar performance to always sending redundantly, but it sends many fewer bytes. Lower confidence intervals offer further reduction in bytes sent, but they also noticeably degrade performance.

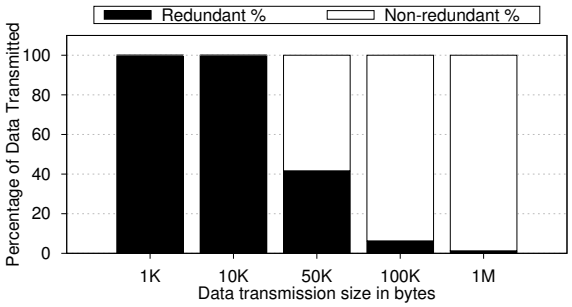


Figure 3.5: Percentage of data sent redundantly and non-redundantly for different data sizes.

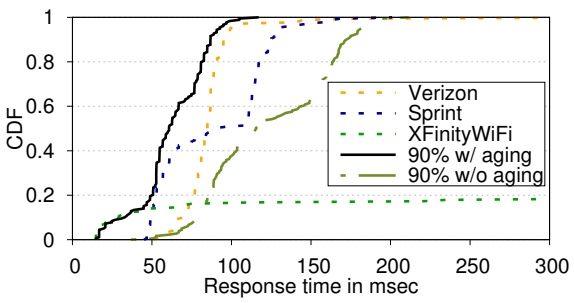


Figure 3.6: Application response time for Yelp in the downtown scenario with and without scaling for sample age.

3.4.5 Effect of mode switching

Unlike prior systems [51, 59], RAVEN does not require applications to declare or hint about data size in order to decide when to transmit redundantly. Instead, it infers small and large transmissions as described in Section 3.3.4. We transmit data from the client to the server 10 times, with a pause of 30 seconds between transmissions, while we replay the D1 trace. We configured RAVEN to always send redundantly in this experiment to isolate the effect of mode switching.

Figure 3.5 shows how mode switching affects the number of bytes sent redundantly for different transfer sizes. We transmit data from the client to the server 10 times, with a pause of 30 seconds between transmissions, while we replay the D1 trace. We configured RAVEN to always send redundantly in this experiment to isolate the effect of mode switching.

For small transfers (1 KB and 10 KB), all data is sent redundantly. For 50 KB transfers, 41.67% of data is sent redundantly. Yet, only 6.28% of 100 KB transfers and 1.27% of 1 MB transfers are sent redundantly. We do not explicitly measure power usage because, in connected cars, the built-in network interfaces are powered by the vehicle engine, and mobile devices connect to these interfaces via the hotspot. With the MPTCP proxy at the AP, mobile devices transmit only a single copy of the data.

We confirmed the micro-benchmark results by examining redundant bytes transmitted for representative mobile workloads while replaying the downtown trace. All applications are unmodified and use our MPTCP proxy. Web, video, and, and application download are three large consumers of mobile bandwidth. We loaded the home pages of the Alexa top 100 Web sites (as of March 2018) in the Chrome Web browser. Overall, 4.9% of bytes were sent redundantly. 2.4% of bytes received and 55.3% of bytes sent by the client are transmitted redundantly. The difference reflects the size disparity between typical HTTP requests and responses, and the results show that our heuristics do a good job of distinguishing small and large transmissions. When we used mplayer to stream a 50 MB mkv video via MPEG-DASH, only 0.03% of bytes were transmitted redundantly. We installed the libreoffice-help-en-us package via `apt-get`, which sent 2.4 MB of data to the client. Only 0.59% of the bytes were sent redundantly. Thus, for heavy consumers of mobile bandwidth (Web, video, and application download), RAVEN transmits very little data redundantly.

3.4.6 Effect of using scaling for sample age

Finally, we examine the impact of RAVEN's policy of discounting samples by the amount of time that has passed since they were collected (discussed in Section 3.3.1). We modified RAVEN to not take sample age into account when calculating confidence intervals. Figure 3.6 compares RAVEN with and without aging for the Yelp application and trace D1 (we chose Yelp because it has the least-frequent network transmissions). The aging mechanism appears to be very important, as RAVEN actually underperforms both cellular networks with the mechanism disabled. Without scaling, RAVEN attaches too much importance to stale samples, creating an overconfidence in its predictions that leads to it not employing redundancy when it should.

3.5 Related Works

Redundancy in mobile computing: DEMS [51] and Meatballs [59] both reduce completion time in multipath communication by transmitting data redundantly. Although both use redundancy to mask prediction uncertainty, RAVEN has several important innovations. First, both prior systems require extensive application hints (DEMS requires applications to disclose transfer sizes, while Meatballs requires applications to distinguish small, latency-sensitive transfers). In contrast, RAVEN requires no hints or application modification. Second, DEMS and Meatballs are user-level implementations. RAVEN's kernel implementation allowed us to add several optimizations that cancel work rendered unnecessary by redundant operations. Third, our evaluation results show substantial benefits available from using three networks. Whereas RAVEN easily scales to any number of networks, DEMS is fundamentally limited to two networks (because each network starts transmitting at one end of the data block and they meet in the middle), and Meatballs computation scales exponentially with the number of networks because it uses joint probability distributions. RAVEN also explores the power of explicit confidence intervals, using them to balance latency and data usage, decide when to transmit redundantly, and test poorly-performing networks for improvement.

Multipath TCP: MPTCP [57] extends TCP to provide multipath communication over multiple underlying networks [122]. Its default scheduler uses the network with the lowest RTT for small transfers and stripes larger transmissions across multiple networks. MPTCP is increasingly being adopted by mobile operating systems, and scheduler design is a very active area of innovation [3, 77]. RAVEN is a new MPTCP scheduler that implements strategic redundancy. Although many vendors are adding support for MPTCP [3, 77], deployment is still not universal. To support unmodified applications and operating systems, we use a TCP-to-MPTCP proxy, similar to the approach used by previous MPTCP research projects [105, 51, 55, 43]. ReMP [43] is

an MPTCP scheduler that transmits data redundantly over *all* available networks; this will consume too much mobile data. In comparison, RAVEN uses redundancy only when it is most likely to improve user-perceived performance for latency-sensitive applications.

Issues of heterogeneous wireless networks: MPTCP performance issues when using heterogeneous wireless networks are well-documented. Several schedulers address aspects of this problem without redundancy. Wischik et al. [146] consider both RTT and congestion to select networks. Yang et al. [151] and Khalili et al. [74] compensate for RTT spikes with adaptive, rate-based scheduling. Paasch et al. [111] mitigate bufferbloat by detecting RTT spikes. While these solutions each ameliorate negative side-effects from unpredictable, high network latency, RAVEN instead attacks the underlying issue by transmitting over additional networks to mask latency spikes in a single network. The limitations of TCP's default RTT estimation algorithm [70] are also well-studied. Several studies [55, 88, 105] show it is vulnerable to RTT fluctuations arising from use of multiple heterogeneous networks. MPTCP has also been used in data centers to aggregate bandwidth of multiple networks [121, 156].

3.6 Discussion

The performance of wireless networks is extremely volatile, especially in vehicular settings. Predicting near-future RTTs of each network is very challenging. Fortunately, modern mobile devices are equipped with multiple heterogeneous network interfaces that offer distinct performance and availability distribution. *Strategic redundancy* improves application-perceiving latency with minimal excessive resource usage. RAVEN is a loadable MPTCP-kernel scheduler module that implements design principles for strategic redundancy without application modification nor significant kernel modification. The evaluation in live experiments shows up to 11x of 95% tail latency improvement, as compared to MPTCP's default scheduler. Therefore, strategic re-

dundancy is beneficial to latency-sensitive applications that need good connectivity to computing infrastructure like the cloud.

CHAPTER IV

Co-locating Computations

Emerging vehicular applications such as driver-assistant and semi-autonomous driving applications need to process large-size inputs collected from vehicles in near real-time. A recent article reports that a vehicle generates 25 GB sensor data per hour [69], equivalent to 7.1 MB per second. The benefit of improved connectivity by redundant transmission diminishes quickly for these data-intensive applications, because large data are striped over multiple networks. The slowest network that blocks server applications to process data, commonly referred as the Head-of-Line (HoL) blocking problem, leads to poor application performance and worse user-experience.

Furthermore, streaming input data to the cloud to process is unreliable due to tail latency and bandwidth hiccups, as discussed in Section 2.1. Also, it is monetarily expensive to ship the data over constrained cellular networks. Upgrading in-vehicle physical computation units can be prohibitively costly and unsustainable for society, apart from reliability concerns, energy constraints, form factors, etc. Fortunately, edge nodes located in ubiquitous cellular towers can provide moderate computing capacity with very low latency (hence, low application response time) and high bandwidth between client application in a vehicle and server computation in an edge node. Telecommunication operators increasingly deploy computations at cellular towers to pre-process user-generated data and to reduce the load on connection to backhaul networks [8, 142, 107]. Note that the current deployment focuses on Software-Defined

Networking (SDN); we envision the computations at cellular towers will become available to broad categories of third-party applications.

Sadly, a distributed nature of edge resources located in "last-mile" hops, i.e., cell towers, makes it challenging to work with emerging vehicular applications. For stationary devices like smartphones and laptops, applications can continue to offload to servers in the same edge node. However, vehicular applications need to migrate or replicate computation servers when a tower switches, which happens frequently and unpredictably in vehicular setups. Reaching back to edge node in the previous tower results in much higher latency than reaching the cloud, as shown in Section 2.2.

The problem becomes complicated for *stateful* applications that must retain a full copy of the application state from the previous node in order to resume the service at the current node. In other words, a stateful application (e.g., object **tracking**) must retain its previous state when the edge node changes. In comparison, stateless application (e.g., object **recognition**) can be replicated across multiple nearby nodes to handle incoming requests. As studied in Section 2.2, inter-tower bandwidth provisioned in current cellular infrastructure is extremely constrained. Moving the state with 1 GB takes an order of seconds, during which client application experiences "downtime" that lags frames or even becomes unresponsive.

Applications can manage and transfer the state when the associated edge node switches for cellular handover; it is unrealistic for the edge (and cellular) providers to expose tower-related information to third parties in the foreseeable future, relying on underlying systems to migrate application state. Existing edge systems [53, 92, 54] accommodate server applications in encapsulated platforms such as VM in QEMU and container in Kubernetes. Because the server application state is encapsulated in a virtual machine, underlying systems can *live migrate* the state in memory without deciphering the structural organization of contents in the application state. Live migration incrementally ships memory states by computing a delta of previously shipped

and current states. When the delta is small enough to ship within a threshold, the migrated server resumes after downtime to ship the final delta.

Live migration works great for applications that have small application states and memory footprints, because the final delta determines application downtime, and the majority of the state can be pre-provisioned to sink node ahead of time, as we explore later in this chapter. To pre-provision a state across future edge-nodes, one needs to *predict* the next edge nodes likely to associate. In this chapter, we first explore a various location prediction methods based on the information gathered in existing cellular infrastructure. The preliminary evaluation of the individual method is disappointing; the joint model offers better prediction accuracy for our target platforms.

Next, we study existing VM migration techniques with two simple stateful applications: vehicle line tracking and a drowsy driving monitor. We measure the application-perceiving downtime with and without optimization in an off-the-shelf VM migration system. The evaluation results are devastating, even with the optimization, resulting in seconds of downtime during which the applications completely become unresponsive. This is particularly problematic for emerging vehicular applications that are *data-intensive*. They process large-size inputs and dirty memory pages rapidly. When the edge node changes, the final delta to transfer is too large to ship within a reasonable time across extremely constrained inter-tower links. As a result, existing migration systems incur high downtime, typically in an order of tens of seconds

We turn our attention to explore a new migration approach that collaborates with applications to identify a necessary portion of application state and hence, to minimize the downtime with a smaller size of state to transfer. Our approach differs two folds from prior works [53, 92, 54] that investigate *compression* approaches to minimize the final delta. First, we acknowledge the fundamental problem in encapsulation and asks the application developers for help, whereas the prior approaches provide a one-fits-all solution. Second, we identify that server applications (serverlets) maintain state across

requests. We introduce *inter-request state* abstraction for an alternative to expose the state without breaking encapsulation platforms. Toward the end, we discuss design choices made along the way building a prototype system, named Croesus as a plugable module. The evaluation results show that Croesus module improves up to 3x in median and 4x in 99% tail latency by consuming 240x less bandwidth to migrate, compared to QEMU's live migration. Finally, the chapter concludes with a discussion of some related works.

4.1 Predicting next location

The location of an edge node that is and/or will be associated with the end-device is critical [133]. When a mobile client moves to a new cell tower, the measurement study of latency in Chapter II tells us that leaving the edge computation behind is a poor choice, because the latency to reaching back to the previous tower is prohibitively high. On the other hand, moving the state *just-in-time* when the new association is detected would take an order of seconds or even minutes to migrate the state from the previous tower because of the constrained inter-tower bandwidth. Therefore, *predictively* migrating at least part of the state in advance to the next cell tower can be highly beneficial.

Unfortunately, accurately predicting the entire future path of a mobile client is very challenging, as noted by previous studies [80, 113, 16, 34, 103]. Unlike smartphones, high-mobility clients can quickly and randomly change their paths in the middle of a trip. Although sophisticated machine learning models [75, 37, 139] can accurately predict the start and end locations (hence, the towers) of a trip based on the devices' history, predicting the exact turn-by-turn routes of the end-to-end trip in advance is extremely difficult.

Fortunately, predictive migration does not need to know every tower along the end-to-end trip in advance, because the state of active edge computations gets modified

quickly and is required to be re-transmitted eventually in the long-term future. In other words, moving the state to towers predicted to be associated more than tens of seconds from now would be wasteful. As a result, predicting near-future locations of the mobile client in next few seconds based on the current location, speed, and trajectory is sufficient. In fact, previous work [115] shows that predicting locations over the next a few seconds is easy based on the trajectory information and the current location of the mobile client, by assuming straight-line movements at constant speed.

Any particular location is typically covered by multiple cellular towers. That means, location prediction alone does not tell which tower the mobile client will associate with. So, given location predictions, we need to identify a set of potential next towers. There are three potential sources of information that might be helpful: a lookup table, coverage dataset, and history of past handoffs. The first, and simplest, is a lookup table that given a latitude and longitude produces a "preferred" tower. The second method makes use of the coverage area dataset. These two are from the cellular service provider; the third is from the mobile client: the history of cell associations at given locations. Unfortunately, we find that none of this information independently is helpful to accurately predict the next tower(s); instead, *jointly* considering them together can give us a useful prediction in the majority of cases.

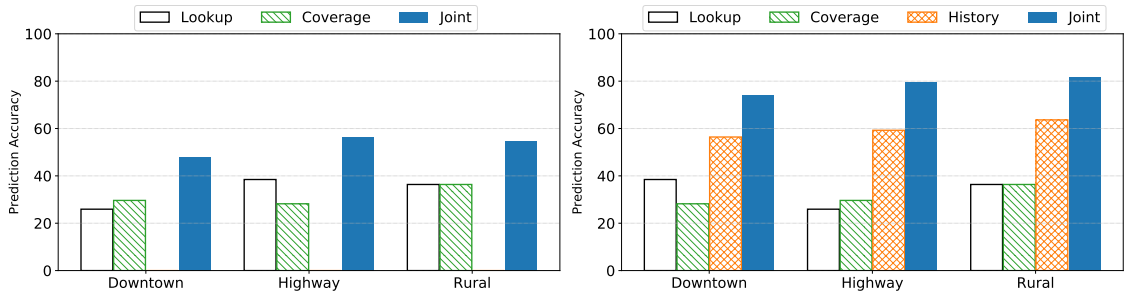
The lookup table was originally created to speculatively choose a tower with which a mobile client in any given location would be associated with the highest probability, using the tower preference map provided by the operator. The table is represented as a geo-spatial dataset of non-overlapping coverage polygons. For our Midwestern state of moderate population density, the total size of the table is approximately 400 MB. However, towers do not need to keep track of all areas all the time; keeping track of only the area nearby would be sufficient.

The coverage data set is similarly structured, though with overlapping coverage polygons, and therefore is both larger and more computationally expensive to use.

While the dataset is almost an order of magnitude larger than the lookup table, towers only need to store the nearby portions of it at any given time. However, even with a small nearby coverage dataset, computing the overlaps is still computationally expensive, because it contains thousands of small, scattered polygons around a small number of primary polygons. So, we refine the dataset to approximate the coverage area of any cell tower with a single shape–a rectangle that turns out to be the most computationally efficient for the spatial database system we have chosen [152]. Because the coverage areas of adjacent towers overlap, a query of this structure typically results in more than one tower. These are weighted probabilistically by distance to the mobile node's location, on the assumption that this models signal strength reasonably accurately. This is likely the best one can do without quite detailed landscape and structure information.

The history data is stored at the mobile client as it moves. This scales linearly in both the number of locations visited and the number of towers seen. However, this is not as punishing as it might seem at first glance, thanks to the fact that most people (and hence their devices) tend to travel to the same places along the same paths repeatedly. In other words, this data structure scales linearly in novelty [126]. When the same location has been associated with more than one tower, we use the most common one. Such an approach does not capture changes in the underlying cellular infrastructure. However, such changes are relatively infrequent, as they often require significant investment. We did not see evidence of this across the three months over which we collected trace data, but we expect that a simple recency bias would be sufficient to capture this phenomenon.

To understand the effectiveness of each of these approaches independently, we examined the location and association traces described in Section 2.2, and subjected them to cross-validation. When history is available, it is the most accurate, predicting the correct next tower two-thirds of the time. The simple lookup table is next,



(a) Without any history.

(b) With one past traversal.

Figure 4.1: Prediction results for tower association changes with four methods with two traversals.

correctly predicting 43% of the next tower associations. The coverage map is correct just under one-third of the time. In sum, in the best case, the prediction accuracy with standalone methods ranges from one third to two thirds out of all towers. In other words, at least one third of towers changed will provide the latency worse than simply relying on the cloud.

When each source of information has benefits and drawbacks jointly considering them altogether can yield higher prediction accuracy. However, treating all the information equally can penalize the most accurate source of information. So, we weight the prediction results by each of the standalone methods based on the empirical prediction accuracy that we computed with the traces we collected in the study section. With the current weights we computed, when all three methods predict different towers, we defer to history when available; otherwise, we make use of the prediction with the lookup table.

We begin by evaluating the prediction accuracy for the next associated tower with the four methods discussed previously. Recall that the first three methods—lookup, coverage, and history—each provide one or more predicted towers. A lookup table maps from latitude and longitude to a presumed preferred tower, and is provided by the cellular operator. The operator also provides a coverage map for each cell tower, which we approximate to reduce computational demand. Finally, the mobile

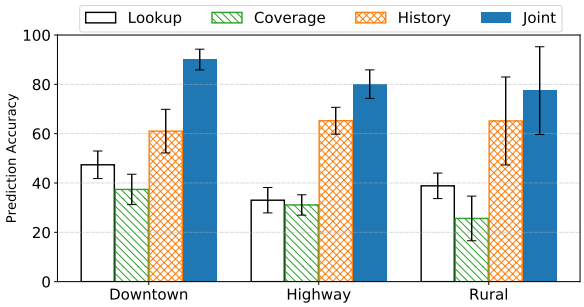


Figure 4.2: Prediction results for tower association changes with four methods with five consecutive traversals. The upper and lower ticks represent the standard deviation errors.

client records past handoffs between towers and supplies that data to estimate the probabilities of next towers, when available. The fourth method combines all of these, by giving weight when individual predictions agree and favoring history when they do not.

Figure 4.1 shows these results for two traversals in each of our three driving scenarios: downtown, highway, and rural. Note that these traversals are different from the ones in the study section. We first compare each of these two traversals, once without the benefit of history, and again with history available. The Y-axis is the percentage of prediction accuracy in each method for all tower handoffs in that traversal. The first traversal without handoff history relies on lookup and coverage methods; with the history from the first traversal, the history method alone outperforms other methods as shown in Figure 4.1b.

Recall that from our study of tower association sequences in Table 2.3, we observed some degree of regularity throughout an individual traversal; we also observed that the regularity quickly fades away across multiple traversals. In fact, none of the sequences was identical to others. That means relying solely on the past history of the same route may often result in a wrong prediction. In our experiments, we see that 36.4-43.6% of predictions based on the past traversal are wrong. In other words, the history may be beneficial but not an oracle. While each method has benefits and disadvantages,

considering predictions *jointly* can mask the wrong predictions and thus, yield a higher prediction. As shown in Figure 4.1b, the joint method significantly improves the accuracy throughout all driving scenarios. Compared to the standalone history method, it predicts 17.7-20.2% more accurately.

We further collected four additional traces while driving the same routes, one after another. The results are shown in Figure 4.2. Clearly, among the standalone methods, the history method is most accurate. The coverage method performs the worst. Especially, in the downtown and rural cases where the radio environments can be dynamically changing, the benefits of static lookup and coverage methods quickly diminish. If a past tower association was due to unusual radio environments, workloads on the towers, etc., the coverage data can likely be more valuable. While any particular standalone methods can be highly accurate, the *joint method* aggregates the benefits of individual methods and yields higher prediction accuracy by 18.9-57.6% compared to the most accurate standalone method in each of the drives.

4.2 Optimization: Hybrid migration

Encapsulated platforms such as VM and container allow migrating stateful services without application-support by shipping a fully encapsulated memory state to the destination node when the end-device switches associations [53, 92, 54]. However, our target data-intensive applications generate dirty memory pages faster than the provisioned inter-tower connectivity can carry. Consequently, the services experience a high downtime to retain a full state prior to service resumptions. In this section, we explore applying a proactive migration technique [27] to reduce downtime, by copying the memory state to likely destination node(s) prior to handovers. Before detailing our findings, we briefly cover proactive and speculative migrations.

A proactive migration copies the full state to destination(s) at the start, and iteratively moves only a dirtied region of the state since the last iteration. In a fully-

proactive approach, the dirty state is incrementally copied as encountered. When the association changes, most of the state is already present at a proactively provisioned tower. Thus, the servers can resume their service shortly after fetching the small residual state at the departure tower, since very little state will be dirtied by continued activity between the early migration and the later association change. The fully-proactive approach reduces the downtime by minimizing the final residual state when the association changes.

Unfortunately, the fully-proactive approach can waste the scarce inter-tower bandwidth by repeatedly transferring a state destined to be unneeded at the next iteration, as large-sized inputs for interactive applications quickly and repeatedly overwrite a huge chunk of state. Therefore, the proactive approach can be quite wasteful. The other end of the spectrum would be speculatively moving the smallest possible state required to resume the service when the mobile client changes association, and then, move the remaining state only as it is needed. The speculative approach consumes the least bandwidth; it potentially incurs significant user-perceiving latency to fetch missing slices of the state at run-time, by reaching back to the old tower.

We have developed a prototype optimized system that employs a *hybrid* of proactive and speculative migrations. Our hypothesis is that if we can proactively transfer *enough* state to the correct destination nodes prior to the handover, a small residual state left in the source node can be shipped *speculatively* after resuming the service. The prototype relies on the joint predictor and uses QEMU's pre- and post-copy modes to employ proactive and speculative migrations. The prototype also estimates total migration time to skip nodes that the end-device will be associated with less than migration time.

To test our hypothesis, we evaluate the prototype with two vision applications commonly used in drives: Eye tracker [36] and AR navigation [7, 6]. These applications are computationally demanding, stateful, and sensitive to latency, as they involve

interactions with users. Before turning our attention to evaluation results, we first give more details about these applications and emulation settings.

The Eye tracker captures frames from a camera facing the driver's eyes at 25 frames per second (fps). These frames are shipped offsite to a server that tracks eye movement, using OpenCV's built-in tracking feature. The server estimates the driver's focus and attention based on not only the new frame, but also a sequence of prior frames. If the driver's eyes are not within a bounding box denoting attention to the road for a particular length of time, the application raises a haptic warning, typically by steering wheel vibration. We measure the time between a frame transmitted and the warning/no-warning response received.

The AR navigation client sends the frames captured from the vehicle's front camera to the navigation service at 10 fps. The server tracks the current lane that vehicle is on and highlights that for the driver. The accuracy of lane detection depends on a sequence of frames. When the vehicle changes lanes, the highlighting moves, giving the driver a visual cue for lane-keeping purposes. The figure of merit for this application is the time between a frame and the highlighting response.

We have evaluated these applications in a testbed that emulates the tower association changes while driving a vehicle. The emulator replays traces of the three driving scenarios described in Section 2.2. Our testbed consists of three machines that emulate the sequence of cell towers. These three take turns by playing the role of the current, the next, and the previous towers. Each machine has 40 cores with 128 GB RAM. To shape the traffic between emulated tower nodes to match the deployed latency and provisioned bandwidth capacity that represents the median inter-tower performance, we use the Linux Traffic Controller (`TC`). We configure the inter-tower latency to 86 ms RTT and the bandwidth to 430 Mbps. Our testbed has two additional machines to emulate a mobile client for tower association changes, and another emulates the cloud with fixed 83 ms RTT for the mobile client.

Application	Trace	System	Median first response time (in msec)	% of response within deadline	Median BW consumption
Eye Tracker	Downtown	Speculative	10,416	46.35	624 MB
		Hybrid	2,093	81.81	703 MB
	Highway	Speculative	9,998	63.61	639 MB
		Hybrid	1,527	86.91	739 MB
	Rural	Speculative	10,755	82.48	451 MB
		Hybrid	1,183	86.67	665 MB
AR Navigation	Downtown	Speculative	11,890	60.08	496 MB
		Hybrid	3,135	86.25	834 MB
	Highway	Speculative	11,681	45.60	528 MB
		Hybrid	3,023	85.34	1.08 GB
	Rural	Speculative	11,614	81.96	523 MB
		Hybrid	3,386	89.02	860 MB

Table 4.1: Median first response time, proportion of responses served within deadline, and median bandwidth consumption per migration for two sample applications over three driving scenarios, with speculative and hybrid migrations.

In this experiment, we measure three metrics. The first metric is the *first response time* after the mobile client changes associated towers. The metric answers the question of when we associate with a new tower, how long does it take before the edge computation delivers the next valid response back to the mobile client? The next metric is the *on-time response proportion*. Each interactive applications has an expected frame rate to deliver a satisfactory user-experience. This metric captures the fraction of frames computed within the deadline implied by that rate. Note that all frames are delivered even if they are too late to preserve consistency between the stateful edge and cloud replicas. The final metric is *bandwidth consumed*, the primary cost of predictive migration. The cost metric suggests the overhead to deliver the dirty but soon-to-be-useless state that is destined to be over-written quickly.

Table 4.1 shows the results with speculative-only and with a hybrid of speculative and proactive migrations. In general, the use of the hybrid migration improves first response time by 5x and 3.5x for tracker and navigation applications, compared to the speculative-only migration. Likewise, hybrid migration meets the response time expectations at least 81% of the time in all situations, while the speculative-only approach can meet as few as half of them. Hybrid migration consumes 13% to 48%

excessive bandwidth consumption in median to improve for the tracker application; it doubles the consumption for the navigation. This is partly because the application generates a dirty but soon-to-be-useless state at the higher rate.

It is imperative to note that although hybrid migration reduces the downtime for edge servers to migrate, the raw downtime numbers are incredibly high, resulting in 1-3 s to resume the services after handovers. In other words, even with optimized migration techniques, the service will be unresponsive frequently when the cellular tower switches. Stalled service is so painful that a service wants to avoid it at any cost [35], even by sacrificing overall latency with using the cloud instead of the edge.

4.3 Inter-request State: segregated yet sufficient

The evaluation results suggest that a full state migration may not be necessary. This lesson is especially *valuable* for the data-intensive applications, because they often maintain a relatively small application-specific state internally. For instance, Augmented Reality (AR) navigation maintains vectors of left and right lines. Eye Tracker needs at the most several previous frames. This leads to two important observations. First, the state that inter-lives across subsequent requests is critical to applications. Second, the inter-request state is tiny, compared to gigantic memory footprints that are inherent residuals of program execution (e.g., static libraries loaded, intermediate outputs). While the inter-request state, i.e., what applications care the most, occupies trivial space in memory, both speculative and hybrid migrations must copy a full memory state to resume the service, wasting precious bandwidth and incurring unnecessarily high downtime.

The small application-specific internal state has motivated us to develop an *inter-request state* (IRS) abstraction, that draws a clear boundary between application-specific portions (e.g., line vectors, previous frames) and process/system-wide sectors (e.g., loaded libraries, system call residuals) in designated memory space in encapsula-

tion. In fact, the idea of state segregation for stateful applications has been thoroughly studied in various domains: system recovery [21, 91, 100], replication [133], and debugging [118, 132, 101]. Unfortunately, we find that it is challenging to apply the existing segregation techniques in an edge-computing environment.

Edge-native applications are increasingly deployed in encapsulated platforms such as VM [54, 53] and container [92]. Encapsulation provides practical benefits in security, resource sharing, etc.; it also obfuscates a boundary between application-specific and system-wide states in memory. This is because encapsulated applications run in a guest operating system (OS). Sadly, without heavily modifying the guest OS, the underlying host OS cannot identify a region of the applications' state. On the contrary, prior segregation systems in other domains [21, 91, 100, 133, 118, 132, 101] have a clear boundary between those states, as applications run in standalone processes. Those systems require developers to annotate variables of interest and install execution tracers to taint-track changes in variables and pin-point memory addresses (e.g., `watch` in GDB).

Fortunately, edge-native applications are servlet applications: they modify stateful variables across subsequent requests and the states remain in edge nodes. That means, tracking changes in the applications' memory heap in run-time may be sufficient to expose IRS abstraction to the host OS. Rather than modifying guest OS's kernel, we can piggyback a small run-time module in encapsulation to trace changes in the applications' heap in memory space. Also, similar to prior systems, variables can be annotated by application developers offline and can be taint-tracked at run-time. The rest of this section describes design choices and implementation details to support IRS abstraction for encapsulated edge-native applications in the context of our prototype module, called Croesus.

```
# global variables
frames = []
lanes = []
```

(a) Initialize variables

```
import croesus
@managed_variables
def app_variables():
    global frames, lanes
    frames = []
    lanes = []
```

(b) Annotate managed variables

```
# RESTful end-point (multi-threaded)
def process(input_frame):
    global frames, lanes
    result = None
    if len(frames) > 0:
      result = track_lane(input_frame, lanes)
    frames.append(input_frame)
    return result
```

(c) Common code block

Figure 4.3: The Croesus module requires 4 lines of code changes to expose variables of interest to IRS managed space. The first four lines of code on the right figure annotates IRS variables. The left two lines are standard practice to initialize global variables that also execute in Croesus version if no previous IRS is detected.

4.3.1 State Annotation

One of the core principles in BumbleBee is that application developers know the best practice to do about potential trade-offs between application fidelity and resource usages. The same principle applies to a case in state annotation in Croesus. Contents in an application's state are inherently tied to intended jobs and features of the application. For instance, an object tracking application can cache all previous frames to achieve the highest tracking accuracy; it can store only the last previous frame to minimize the memory footprint. Therefore, only application developers can pinpoint a portion of state in memory critical for intended application features.

The Croesus module can taint-track variables shared and modified across requests. Requests themselves are stateless. Any *global* state persists in a form of global variables. Applications written in modern programming languages (e.g., C++ and

Python) declare global variables at initialization and keep an application-specific state in those global variables at run-time. When migrating, the Croesus module can move only those variables, leaving a majority of bytes in memory behind (e.g., loaded libraries, networking residuals). The module can restore the state at destination by loading migrated variables in the application's heap. Unfortunately, the module must re-launch the application when restoring. The global variable initialization is a part of an application execution path by default. As discussed later, the Croesus module skips the global variable initialization when restoring a transferred state.

The Croesus module supports a simple annotation to include variables in managed IRS space in memory. Figure 4.3 shows a code snippet in a lane tracking application that we evaluate later. Application developers write a new function with annotation to initialize global variables that change across inter-requests, shown in Figure 4.3b. Unlike prior systems that require custom programming syntax, the Croesus module integrates seamlessly with the existing programming model. Only changes to benefit from IRS are shown on the right side; the rest of the application source code remains intact, as shown in the Figure 4.3c. Therefore, the Croesus module allows application developers to pinpoint variables to be included in managed IRS space.

4.3.2 Managing an Inter-request State

The programming model remains unchanged, but the consistency of inter-request variables may not. In order to ensure a consistent state of variables in IRS across requests, the Croesus module needs to taint-track changes in variables per request. The problem is that a migration can start while processing request(s), resulting in an inconsistent state of the variables. A single-thread server that processes requests sequentially is less problematic, because at most one request needs to be completed before migration starts. Sadly, a single-thread server imposes such high latency for edge-native applications that inputs are sent at fast speed. If processing old request fails to finish before a new request is received, not only is the migration delayed

to wait for completion of the later request, but also a queuing delay is inevitable to process requests sequentially. So, edge-native applications are typically multi-threaded servers, and that multiple requests can modify the global variables concurrently.

Application developers can implement a lock to ensure consistency across concurrent requests; a migration can start before outstanding requests are processed. Abandoning outstanding requests can accelerate migration waiting time. However, without a cautionary signal back to the client to retry, the variables become inconsistent as the outstanding requests are permanently lost when abandoned at the source node. In fact, a source node can have a state that reflects more requests processed than the one in destination node when migration completes before outstanding requests are processed.

On the other hand, waiting indefinitely for the outstanding requests to complete can result in a significant increase in the migration waiting time. The client sends inputs at fast speed such that at least one request is in the middle of processing and typically is incomplete before later requests are received. Therefore, the Croesus module needs to make an adequate decision to abandon incoming requests and to wait for outstanding requests to process before migration starts.

Another prime overhead to consider comes from IRS commit. The Croesus module can taint-track changes in variables across subsequent requests; the overhead of taint-tracking per request can be high. This is because the module needs to commit in two phases for a consistent state across inter-requests: one when a request is received, and another before a corresponding response is sent back to the client. The simplest way to maintain consistency is to commit managed variables to a persistent storage per request, and defer new requests until a response is returned. Recall that the motivation for IRS is to reduce the total migration time. Saving to storage is simple to implement and easy to migrate, as we only need to migrate the latest state file. But it can incur high latency overhead for applications with a large-sized state (e.g.,

a lane departure application that needs frames during the past 10 s), because the response is returned after commit.

Incremental commit is an alternative technique we have considered. Instead of saving all bytes in a managed space, we commit only the modified and additional bytes observed. This approach may offer savings in storage resources; the total number of bytes to transfer for state migration may remain unchanged. The approach can complicate a state dependency resolution when restoring at a destination node. The next approach considered is asynchronous commit. It reduces the latency overhead for commits by having a dedicated thread to commit changes. Unfortunately, we find that for data-intensive applications that send input data at a rapid pace, the commits build up a large backlog quickly. By the migration time, the latest commits are still waiting to be transferred, resulting in merely moving the overhead to the migration stage.

Instead of copying contents of variables per request, the Croesus module uses a shallow commit approach. The module maintains a shallow pointer to memory addresses of the variables' contents. Maintaining pointers prevents the variables from being garbage collected. This shallow commit approach is possible because the modern application frameworks change pointer addresses only when contents are modified. The Croesus module keeps a list of pointers per request and response pair. When migration is requested, the module installs a watchdog to observe a number of outstanding requests to complete. Then, the module waits for the number to drop to zero. It defers processing forthcoming requests received after the migration request by putting into sleep when requests are received and inspected. Instead of dropping requests immediately, we choose to defer requests to share a consistent view on processed requests with the client. The deferral mechanism allows the module to trigger a connection to timeout and signal back to the client that a request has failed to be processed. The client can keep track of a progression of successful requests and retry

failed requests later when the migration completes. Note that the Croesus module creates a sequence number for each request and appends it to the response's header when it is returned for successful and failed requests. Thus, the client can use the sequence number to track progression of successful requests, rather than maintaining its own.

A deferral mechanism relies on client-server applications to maintain a correct ordering, which is a reasonable assumption for a transient mobile networking environment. Applications that value less stale old requests benefit from faster migration completion time with deferral. However, orders of requests can be shuffled. In other words, future frames can be processed earlier than previous frames that are retransmitted for connection timeout. This can be problematic for applications where the state is heavily dictated by the orders of input requests (e.g., LiDAR 3D mapping). Those applications can benefit from the Croesus module's Buffer-To-Forward (BTF) mechanism, that buffers incoming requests received at source node during migration and later forwarded to sink node. The module at destination buffers incoming requests to process forwarded requests first. As a result, new requests suffer temporal latency overhead until forwarded requests are processed first. The BTF option guarantees in-order processing of requests in accordance with received orders. By default, The BTF is disabled in the current prototype module.

Once the number of outstanding requests drops to zero, the Croesus module installs a lock on managed variables in IRS. Then, it copies actual contents from memory addresses obtained from taint-tracking variables. It serializes and sends the state to the destination node.

To sum up, the Croesus module ensures a consistent state for multi-thread edge servers so that multiple requests concurrently modify an application-specific state. It uses a shallow commit to taint-track changes in the variables to minimize commit overhead. Once migration starts, the module waits for outstanding requests to com-

plete; it defers later requests received. The deferral allows the signal back to clients to share a consistent view of processed requests to retry later, which is a natural recovery method in a transient mobile networking environment like edge-computing. The module also provides the BTF option to ensure in-order processing of received requests.

4.3.3 Optimization for State Restoration

Recall that live VM migration ships a full memory state. It allows migrated VM to resume simultaneously as required components are already loaded in the sink node's memory (e.g., libraries, application binary). In comparison, the Croesus module reduces migration time by shipping necessary IRS in memory, leaving other run-time contents including the encapsulation platform, i.e., VM, in source node. That means the Croesus module must spawn a fresh VM that the server application can resume with the received state.

Unfortunately, VM booting time can be very long, in an order of seconds [72]. We optimize restore time with a lightweight VM snapshot. We pre-provision a snapshot image for each application in a fully booted state. The snapshot includes necessary libraries and execution binaries for the application to start; it has no application server running yet. When migration to sink node completes, the Croesus module at source node invokes RPC to spawn a new VM instance from a snapshot image and to start the application server with the Croesus module loaded in VM. To reduce copying overhead from host to VM, the received IRS state is shared with the module in VM through a Virtual socket (vSocket) interface.

To restore the application's state correctly in VM, the Croesus module starts the application server from scratch to load required run-time components in sink node. The module injects a watch-point for initialization of managed variables, similar to taint-tracking. When the server initializes managed variables included in IRS, the module overrides pointers to the received state. The module also injects `JUMP`

instruction at the end of the initialization function to skip re-initializing managed variables. Once contents of the variables are loaded from the received IRS, it disables watch-point and installs a tracer for taint-tracking for the future migration.

4.3.4 Implementation

The prototype of the Croesus module is implemented in Python v3.9 to commit, to migrate, and to restore managed variables in IRS. It uses Python's default pickle library to (de-)/serialize variable objects sent and received over the network, supporting user-defined data-structure as well. Application developers annotate a function that initializes variables to be included in IRS space, as shown in Figure 4.3b. At the run-time, the module installs global and local tracers [117] in the application's server execution stack. The tracers allow the module to track variables in the application's runtime stack when a request is received and a response is sent. The module evaluates changes in managed variables when the next line of execution is return statement. The module looks up and marks the most recent request received to taint-track changes in managed variables. To keep the variables consistent, the Croesus module implements a global lock when shallow pointers are recorded for changes in managed variables.

4.4 Evaluation

To evaluate the efficacy of the Croesus module, we seek answers to the following questions:

- Does Croesus allow fast migration?
- Does Croesus's IRS abstraction impose overhead?

To answer the first question, we measure three prime metrics. First, we measure the end-to-end latency for input requests. The latency includes not only processing time but also transmission delay for wrong tower association and potentially Croesus's overhead. Second, we measure the bandwidth consumption to transfer the managed

state in IRS abstraction. As discussed in Section 2.2, the inter-tower bandwidth is prohibitively constrained in comparison to a VM's full state to migrate. Lastly, we measure application downtime, the uttermost metric for user-facing applications. Note that downtime differs from latency. Downtime reflects the duration of unresponsive applications for migration delay, whereas high latency can result in the hands-of-wavering delay in the request being processed. Lastly, we run micro-benchmarks to understand the potential overhead incurred from maintaining Croesus's IRS abstraction.

4.4.1 Methodology

We evaluate our Croesus prototype and the QEMU migration system with one synthetic and two realistic vehicular applications. The synthetic client application sends a time-stamp to the server in an edge node. The server returns an MD5 hash key, computed from previous and current time-stamps. We measure the latency between when the time stamp is sent and the key is returned. The synthetic application has lightweight computation and a minimal memory footprint from the small payload and intermediate result.

We extend the evaluation with AR navigation from the previous evaluation. The AR navigation application requires the previous frame to detect lane changes, as described in Section 4.2. The application with Croesus places the frame in IRS. Next, we further evaluate the LiDAR-based object tracking application. Unlike stereo camera-based applications, LiDAR-based applications generate much more input data, called point-clouds. The client sends point-clouds to the processing server at the edge. The server employs ResNET model to detect objects present in the LiDAR inputs, identifies objects also present from previous inputs, and sends back tracking results. The server maintains past one LiDAR input to track new and old objects. We use Kitti dataset [47] to emulate the client vehicle. Each reading contains at least 100,000 sample points of x, y, z, and depth measures. Each reading is on the average 1.7 MB,

which is an order of magnitude more than stereo camera input.

We evaluate the applications in a trace-drive emulation. Unlike a mobile networking trace-driven emulation [79] that requires at most three machines—client, server, and possibly router, the edge-tower emulation may require as many machines to emulate towers observed in collected traces. Cellular towers are already extensively ubiquitous in reasonably populated locations such as Ann Arbor; rural areas and even highways are also typically covered redundantly by many macro and micro cells. Redundancy in tower coverage is essential to provide highly available cellular service.

We have used three machines to emulate edge-towers. Each machine has 8 cores and 32 GB RAM. We have used another machine to emulate a moving vehicle. The traces contain a sequence, duration, and the ids of towers associated during collection. Throughout the experiments, the client sends input requests to only one edge-tower. This is a reasonable assumption to ensure a consistent state at server-side. We have built a trace replayer that injects `iptable` rules to redirect requests to a machine that emulates the currently associated tower in the trace. Instead of the client looking up tower-to-machine mapping, the client always sends to a local address where the replayer has configured redirection rules to the server machine according to the traces replayed. If the currently emulated tower differs from the currently associated tower, the replayer injects 86 ms RTT to the tower machine.

The replayer announces a change of tower association to the controller that issues a migration decision. We have implemented a simple controller that always issues migration decisions to the newly associated tower to match with an existing just-in-time migration system [54]. The controller sends a migration request to the replayer that redirects requests to a machine that currently serves the client. When successful, the controller is notified and sends a switch RPC that adds a new redirection rule for the replayer. The migration is canceled when the tower association changes preemptively before the migration is completed. The replayer notifies the controller about the failed

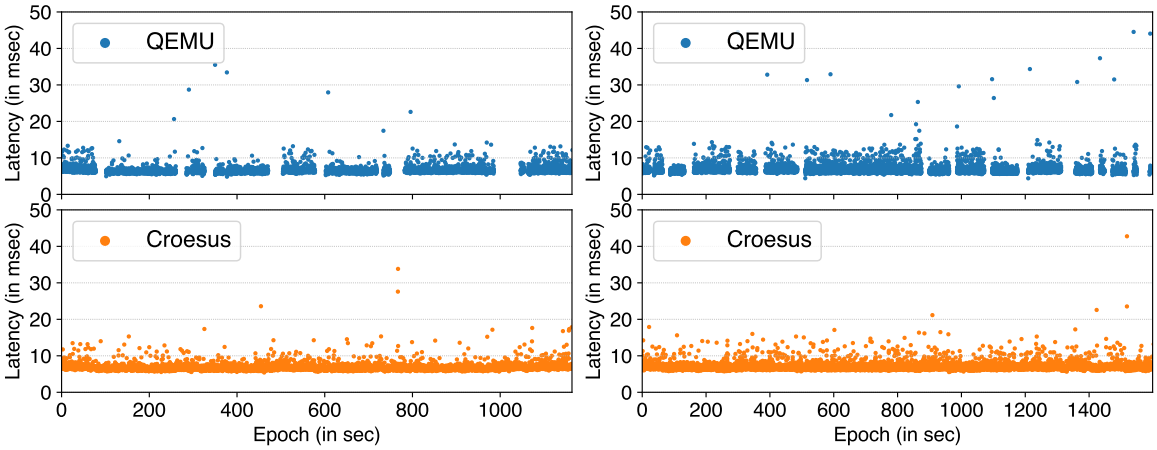
case to share at which tower the server runs.

We configure the client to install a connection timeout at 1 s. The client registers callbacks for a request's success and failure. It retries failed requests that are less than 3 s old. We effectively allow the client to send a request three times at most. We choose a 3 s threshold because the re-transmission policy in Linux is also three times. A callback for successful requests is invoked with response from the server that contains the result of the request processed and the time-stamp when the request is first generated and sent. For bandwidth consumption, we log the total number of bytes sent over a network from the source node. Bandwidth consumption also includes a canceled migration case that a tower association changes preemptively before migration completes. We have modified QEMU to expose the total bytes transferred.

4.4.2 Results of Trace-driven Emulation

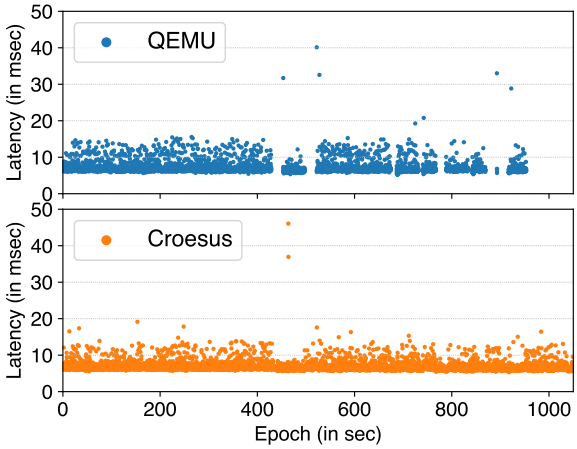
We begin by evaluating a synthetic application that sends and receives small payloads with lightweight hashing computation. The application generates a small memory footprint to migrate, as inputs and outputs are only several bytes and computation is lightweight. Figure 4.4 shows the application-perceiving latency by the synthetic application across three traces replayed. Figure 4.4c shows the rural scenario, where QEMU, shown in blue dots, delivers latency between 8 and 17 ms for the first 400 s that the client stays with the initial tower. However, when switched to a new tower, the application encapsulated in QEMU experiences downtime for approximately 15 s. The downtime continues to occur when the associated tower switches. The rural trace has the least number of tower switches. In comparison, Croesus, shown in orange dots, consistently delivers latency below 20 ms; it also suffers from a temporal high latency to transfer the state. Because the state encapsulated in IRS is small, Croesus migrates the state with negligible downtime.

QEMU suffers from painful frequent downtime in a highway scenario that has the most number of tower changes. Figure 4.4b shows that QEMU encapsulation results



(a) Synthetic downtown trace

(b) Synthetic highway trace

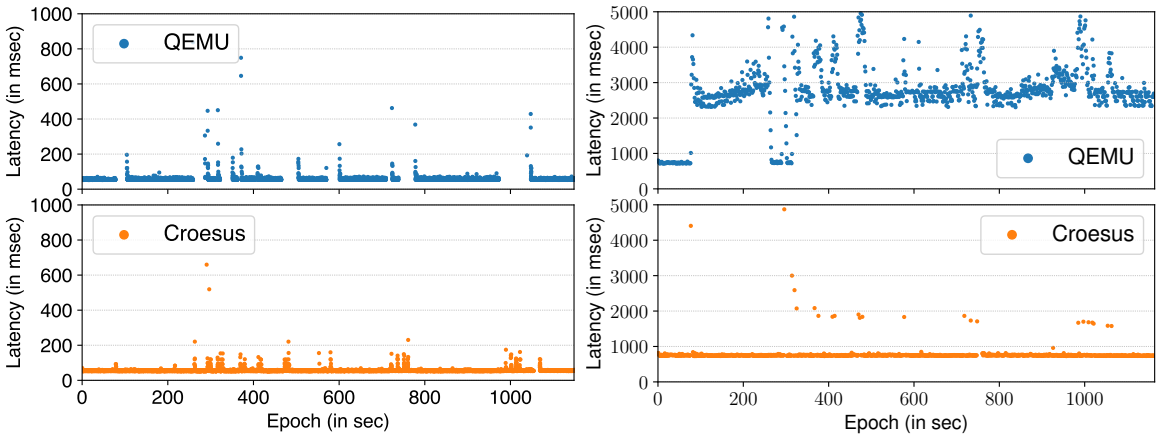


(c) Synthetic app with rural trace

Figure 4.4: Results of the synthetic application's response with QEMU and Croesus across three traces replayed.

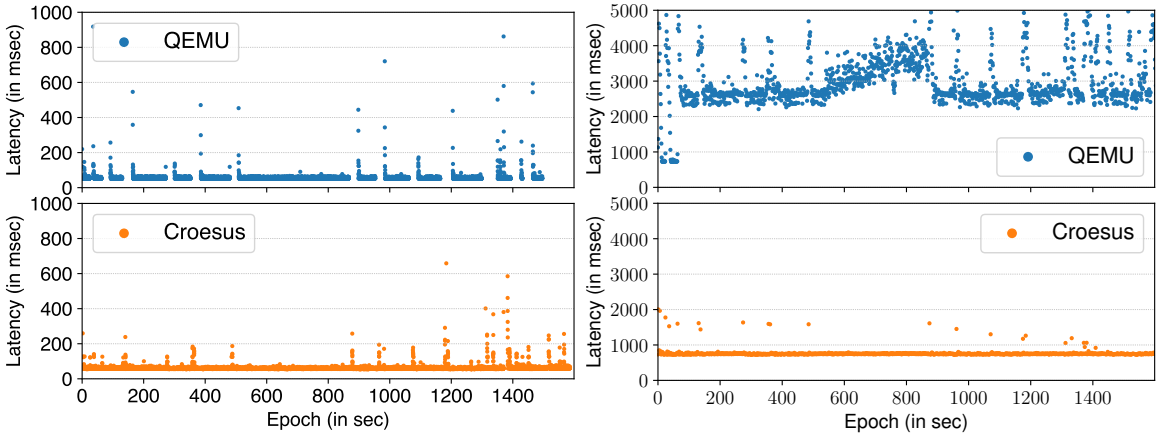
in 14 instances of downtime lasting more than a second. On the other hand, Croesus consistently delivers latency with no downtime more than a second. Even for a moderate case of downtime scenario, QEMU clearly experiences from high downtime lasting more than ten seconds, as shown in Figure 4.4a. Despite the lightweight computation and small memory footprint of the application, QEMU retains a full memory state to migrate. We suspect that this is because multiple memory copies from deep down in the kernel to high in the application layer dirty the memory pages to transfer, resulting in a large state to transfer for QEMU.

Figure 4.5 shows the results for AR navigation on the left and LiDAR object



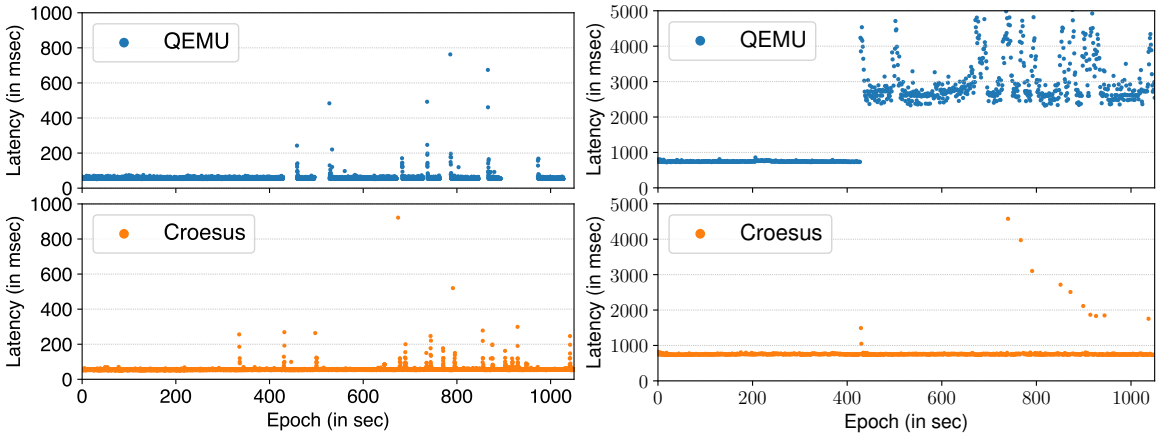
(a) AR app in downtown trace

(b) LiDAR app in downtown trace



(c) AR app in highway trace

(d) LiDAR app in highway trace



(e) AR app in rural trace

(f) LiDAR app in rural trace

Figure 4.5: Results of AR navigation and LiDAR applications' response with QEMU and Croesus across three traces replayed.

tracking applications on the right. Across three traces, AR navigation exhibits similar results to the synthetic application with QEMU: frequent, disjointed downtime

clusters when the tower switches. Figure 4.5e shows a rural scenario. Clearly, QEMU incurs downtime lasting more than a second. Worse, QEMU has a long downtime period between 850 and 900 s epochs. We identified from the log that QEMU cancels migration when the tower switches before migration completes and instead issues a new migration request for 7 times. Consequences of canceled migration by QEMU are also noticeable in other scenarios: after 1500 s in Figure 4.5c and around 1000 s in Figure 4.5a. In comparison, Croesus briefly incurs high latency but no downtime across all scenarios, benefiting from the small size IRS. The results are expected as AR navigation has large payloads that dirty memory pages rapidly, even though the application-specific state is several hundred bytes. In other words, QEMU migrates destined-overwritten memory pages repeatedly, because it cannot distinguish the memory state necessary for the application.

Interestingly, the results of the LiDAR application are significantly different from the other two applications, especially for QEMU. Figure 4.5f shows results of the LiDAR application in a rural scenario. Up to the 420 s epoch, both QEMU and Croesus exhibit latency around 700 ms. After the client departs from the initial tower, QEMU never retains the latency of the previous level. From the log, we found that this is because the QEMU migration never converges. Recall that QEMU requires transferring the final delta of state to resume the service in the sink tower. Sadly, the LiDAR application generates a large final delta for two reasons. First, the payload for input is large, an order of magnitude larger than AR navigation. We intentionally configured the input interval to 1 second. In other words, input bytes per second is comparable to AR navigation. Secondly, the application employs a neural-network model ResNet, that creates many temporal, intermediate variables per input. Consequently, these intermediate variables occupy and dirty large portion of memory space in VM, resulting in incomplete migration, aligning with observations in previous work [92]. Additionally, Figure 4.5b shows sporadic latency hiccups at epoch 300 s, as the client switches

App	Trace	System	Latency				Cumulative Downtime	Bandwidth Consumption
			Mean	Median	95%	99%		
Synthetic	Downtown	QEMU	344.91	5.04	2212.32	7249.60	102.90s	11.12 GB
		Croesus	7.17	6.52	7.57	9.80	1.40s	11.04 KB
	Highway	QEMU	419.83	5.16	2568.90	7404.53	213.80s	23.03 GB
		Croesus	7.66	6.66	7.77	10.53	2.10s	16.38 KB
	Rural	QEMU	232.88	5.26	1296.80	7092.04	65.40s	3.95 GB
		Croesus	6.65	6.41	8.12	11.66	1.20s	6.29 KB
AR Navigation	Downtown	QEMU	106.27	53.07	62.98	2125.26	284.00s	13.36 GB
		Croesus	116.66	53.78	74.96	2159.20	26.00s	404.22 MB
	Highway	QEMU	107.42	51.62	61.02	2139.59	401.90s	20.92 GB
		Croesus	105.82	59.88	69.13	2176.50	45.80s	426.57 MB
	Rural	QEMU	92.94	53.24	62.80	1294.24	227.10s	10.81 GB
		Croesus	104.31	54.04	60.88	2178.10	16.20s	255.58 MB
LiDAR Object Tracking	Downtown	QEMU	2677.61	2713.58	4067.22	4817.55	0s	32.89 GB
		Croesus	778.50	747.00	778.69	1834.14	55.00s	220.01 MB
	Highway	QEMU	2942.04	2694.79	4318.64	4975.41	3.00s	73.68 GB
		Croesus	760.64	746.60	769.81	1351.28	43.00s	321.55 MB
	Rural	QEMU	2115.83	2571.64	4205.82	4834.09	0s	29.58 GB
		Croesus	770.87	750.76	775.35	1118.63	29.00s	126.93 MB

Table 4.2: Mean, Median, 95%, and 99% application-perceiving latency for synthetic, AR navigation, and LiDAR Object Tracking applications. All values are in msecs, unless explicitly noted.

between initial and new towers back and forth. Figure 4.5d shows a noticeable queuing delay between 500 and 900 s. In comparison, Croesus consistently retains low latency, except for a temporal latency spike during migration.

Table 4.2 summarizes the application-perceiving latency, cumulative downtime experienced, and bandwidth consumption across three traces for each application with QEMU and Croesus. Interestingly, Croesus rarely improves mean and tail latency for AR navigation compared to QEMU. For instance, in the rural case, Croesus delivers slightly worse mean and 99% tail latency compared to QEMU. However, Croesus significantly improves cumulative downtime by 14x in the rural case. This is contradictory with the LiDAR application. Croesus incurs 29-55 s downtime throughout three traces whereas QEMU incurs 0-3 s downtime. On the other hand, QEMU delivers much high latency across all traces. This is because QEMU seldom completes migration for the LiDAR application and thus continues to offload to the server located in the initial tower, resulting in 3-4x median latency and up to 3x in 99% tail latency, compared to Croesus.

In bandwidth consumption, the benefit of a small segregated state is clear. For the

synthetic application, QEMU consumes approximately 4-11 GB bandwidth throughout three traces. This is expected because QEMU transfers all of VM's memory pages to resume the services, even though the majority of memory pages are hardly used by encapsulated applications. In contrast, Croesus transfers only the application-specific state denoted in IRS as described in Section 4.3.1, resulting in a 6 order of magnitude bandwidth savings. For LiDAR application, QEMU utterly wastes precious bandwidth. In the highway case, QEMU consumes approximately 73.68 GB but completes the migration only once. On the contrary, Croesus consumes less than 2% bandwidth but consistently delivers low latency.

4.4.3 Commit Overhead Micro-benchmark

Migration is composed of three phases: commit, state transfer, and restore. The motivation of IRS is to reduce state transfer time by transferring managed variables in a small IRS space. However, the benefit can quickly diminish even for an IRS sized smaller than the available bandwidth, when the overheads to commit and restore are substantial. So, we measure the overhead to complete the end-to-end migration stage by repeating the migration for different sizes of IRS from 53 KB to 53 MB for 100 times each. We breakdown the time into two: one for commit and another for state transfer and restore. In the commit phase, the Croesus module copies and serializes managed variables pointed to by shallow pointers captured at every request completion. The time to complete transfer phase is dictated by the available bandwidth and size of the IRS to transfer. In the restore phase, the Croesus module deserializes and changes the managed variable's pointers to received ones.

For each request, the Croesus module acquires a global lock that prevents other threads from changing managed variables' pointers to maintain a consistent state. The lock can impose a significant overhead for multi-threaded applications because concurrent requests need to wait for the lock to be released. For high input cases, the latency overhead can be significant. The module optimizes the overhead by maintain-

Frames per second	Average	Median	90%	95%	99%
5	10.85	10.73	13.44	13.75	16.75
10	11.30	10.62	13.42	13.86	21.66
25	10.49	10.59	12.76	13.85	18.70
40	12.53	10.70	16.50	17.86	29.26
60	11.81	10.23	13.99	16.18	54.94

Table 4.3: Average, Median, 90%, 95%, and 99% latency at different frames per second. All values are in usecs.

ing only a shallow copy of variables, i.e., pointers to bytes, to avoid being garbage collected. Table 4.3 shows the latency with optimization when frames are received at a different frames-per-second rate. Overall median latency is around 50 usec. For the high input rate, 95% tail latency ranges from 13 to 18 usec. The result shows that shallow-copy optimization imposes negligible latency even with a global lock for state modification.

4.5 Related Works

Provisioning Edge Computations: Prior works [54, 53, 92, 108, 44, 133] have carefully considered *how to migrate* stateful edge computations, but have paid less attention to *where* such migrations should be located. Unfortunately, these systems may perform poorly when location choice is inaccurate. One approach is to migrate *on-demand* [54, 53, 92, 133], but such approaches would suffer long pause times during migration over the cellular infrastructure that we consider here. Alternatively, other systems rely on an oracle to predict when and where to migrate [44, 108, 82]. In contrast, Croesus considers where, when, and how to perform a migration to minimize client disruption, preserve low-latency response times, and conserve scarce bandwidth between cell towers and the infrastructure.

Just-in-time provisioning [54] focuses on reducing the time it takes to reconstruct a baseline image of edge computations, when the mobile client moves. The system distributes the common baseline image to all edge nodes, and ships the delta of

custom images when requested. Later, the same group proposes using compression on the delta to minimize the migration state [53]. While both approaches can be useful to reduces the size of a dirty state to ship, they employ a *responsive* migration approach, rather than a *predictive* one, and they perform poorly when the mobile clients change edge nodes frequently. The systems incur the downtime of edge computations in an order of seconds to even minutes, which is unacceptable performance to latency-sensitive interactive applications.

Service handoff [92] periodically checkpoints the active edge computations and transfers only the modified state to the potential next edge site, similar to Croesus's speculative migration. Unlike Croesus, the service handoff does not defer migration in order to potentially save tower-to-backhaul bandwidth—a constrained resource in current cellular deployments. Slingshot [133] employs a proxy-based approach to replicate the edge computations, similar to our proxy at the backhaul. However, Slingshot relies on an oracle to locate the close edge computations, which can incur a significant overhead when communicating with *nearby* edge sites and impose high latency, as we have shown in the study section.

Shadow [44] employs the redundant deterministic execution of the same edge computations on multiple nodes. Similarly, our system employs redundant execution to the cloud as an ultimate stateless replica in order to provide a reasonable latency to users when the edge computation migration induces high latency. However, Shadow migrates point-to-point, unlike our multiplexing approach from the backhaul, to save excessive bandwidth consumption, and requires full logs of I/O and intermediate states that can be larger than the state in memory for interactive applications. Steel [108] uses the cloud as an ultimate backup location to accommodate latency-tolerant applications while seamlessly migrating counterpart applications to the edge. This is similar to our mechanism that falls back to the cloud when predictions are incorrect or associations are too short for migration to be helpful.

Mobility Prediction: Many prior works [80, 16, 34, 90, 103, 113] investigate the degree of prediction accuracy using the mobile device's historical data. Most recently, Lee et al. [80] conducted a measurement study that shows frequent handoffs from cell to cell when mobile clients are in low and intermediate moving speed. The authors report the average handoffs of 26 cells for short drives, which align with the study we conducted. The authors propose a system that predicts a near-future uplink throughput, which differs from the objective of Croesus. Both Deshpande et al. [34] and Nicholson et al. [103] use the mobile's past sequence of WiFi APs and locations to estimate the next association. This takes advantage of the fact that people tend to repeat similar routes; our approach depends on the same fundamental idea. However, we expect that the number of cells that can potentially serve a location are more numerous than the number of WiFi APs might be, so there is some reason to think the former is more difficult. Furthermore, we also take into account the information provided by a cellular operator about the preference for and coverage of towers for each location.

CAPS [113] infers full-path routes from past cell association sequences, similar to our joint prediction method that partially uses history when available. CAPS requires the entire past association to infer full-path routes. In comparison, Croesus only needs to know the next many seconds of a path, as the objective function is to predict the next migration or two. This requirement simplifies the complexity of Croesus. Becker et al. [16] deduce partial routes from past cell handoff traces. The authors argue that longitudinal handoff patterns are stable on prior wireless technology; our longitudinal measurements on the up-to-date cellular infrastructure suggest that handoff sequences are hard to predict. Hierarchical Location Prediction (HLP) [90] combines the user's previous cell association history and the mobile node's current trajectory, much as our approach does. However, as with other prior systems, our approach differs in also considering the operator's tower and coverage information.

VM Migration and Control Techniques: Migration has been well-studied in the cloud environment. [27, 62, 63, 112, 147]. The distributed nature of edge computing complicates the problem. Clark et al. [27] introduce the pre-copy approach of live VM migration; our system is built on this mechanism. Hines et al. [62, 63] propose a hybrid migration approach that employs both pre-copy and post-copy migration techniques, similar to our hybrid migration scheme. Croesus adds the consideration of client mobility, deferring the start to save bandwidth.

Sandpiper [147] tackles the decision of when to migrate. It trades application performance for bandwidth consumption, similar to skipping short-lived towers in Croesus. Sandpiper decides where to migrate based on the load of a candidate, whereas Croesus decides the next edge tower(s) based on the mobile node's location and cellular coverage information. AutoControl [112] proposes adaptive migration that explicitly considers the potential application performance improvement by migration. However, this is based on workload, not client mobility; the latter is the focus of Croesus.

Application State Segregation: The idea of state segregation has been thoroughly explored in system recovery [21, 91, 100] and debugging [118, 132, 101]. Lowell et al. [91] investigate these on general principles of failure recovery for applications. The authors propose the Save-work invariant failure-recovery mechanism, that requires applications to explicitly commit the application state to persistent storage, similar to commit in Croesus's IRS abstraction. Similarly, Microreboot [21] accelerates state recovery for software failures by saving the application state across distributed data storage for consistency. Whole-system persistence [100] leverages non-volatile memory to accelerate application state recovery through suspend and resume events, similar to Croesus's recovery mechanism.

Software debugging is another prime domain that utilizes state segregation. Bugnet [101] checkpoints application state by recording the input to applications

through load instructions, omitting the system-wide state. Likewise, Croesus intercepts input messages to ensure a consistent application state. Rx [118] introduces rollback-recovery that preserves a checkpoint application state in memory to accelerate state reconstruction for debugging. Flashback [132] records changes in an application state through local computation proxy facility, called shadow process, similar to Croesus's modular design.

4.6 Discussion

Emerging vehicular applications can potentially benefit from ubiquitous cellular edge nodes, that provide very low latency and moderate computation capacity. Unfortunately, the distributed nature of edge nodes requires migrating stateful server computation when the vehicle moves and switches associated edge node. Existing migration systems use a full state migration, that moves entire memory bytes from one node to another. Full migration is required because the underlying system cannot distinguish between application-specific and encapsulation-wide states such as kernel and libraries. We propose an IRS abstraction that segregates an application-specific state in a designated memory sector with only four lines of code changes in the application-code base. Our prototype module, called Croesus, maintains a consistent state across subsequent requests via background commit by a watchdog. The evaluation results with synthetic application show that the Croesus module improves cumulative downtime by two orders of magnitude. The results with two realistic vehicular applications are surprising: compared to QEMU, Croesus delivers 3x and 4x improvements for median latency and 99% tail latency, respectively, while using six orders of magnitude less bandwidth, a precious resource in constrained cellular networks.

CHAPTER V

In-network Application-aware Adaptation

Distributed edge nodes improve application connectivity by delivering extended computing power in very low latency. Despite the improved connectivity to edges, some applications must interact with other services spanning across both edges and clouds. To leverage both infrastructures, application developers increasingly employ hybrid application architecture that spans across edges and clouds [96, 45]. Hybrid application architecture also provides a number of benefits: to lower costs to distribute workloads, to build services resilient from network-failures, for better performance, etc. Unfortunately, hybrid architectural applications communicate across fundamentally turbulent network environments.

This limitation is acute for hybrid applications, because unlike within a single device, cluster, and data-center (DC), network conditions between the edge and the cloud can change unexpectedly [107, 23, 150], and data partitions are not uncommon [9, 2, 99, 10, 48, 49]. Even within data-center networking, Section 2.3 shows high bandwidth volatility for inter- and intra-DC connections. When network conditions degrade, it is crucial for applications to adapt their internal behavior in response. For example, a machine-learning (ML) application may switch to an edge inference model with lower accuracy to compensate for higher network latency, and a stream-processing application may aggregate more aggressively to compensate for a drop in inter- and intra-DC network bandwidth. In these cases and many others, *application-*

aware adaptation [106] is the key to maintaining acceptable quality when network conditions degrade.

Unfortunately, applying application-aware adaptation to hybrid application architecture is onerous. At the application level, adaptation strategies are often tightly coupled with another functionality in a single application, such as a video-processing application that implements adaptive bit-rate logic and video transcoding. As a result, fine-tuning or modifying an application's adaptive behavior can require changes to a large codebase that is often maintained by a separate development team. At the network-transport level application-oblivious responses to variable network conditions, such as TCP congestion control, provide fair bandwidth allocation, but only the application knows how to change its internal behavior as conditions change. Thus, providing a lightweight *in-network* processing facility for application-aware adaptation is crucial to fill this gap in between.

This chapter describes a journey in building a lightweight in-network facility for application-aware adaptation. It starts with a background discussion of building block systems, focusing on micro-service orchestration. Next, it enumerates the problems in existing systems that govern design principles. Then, it elaborates design and implementation of a prototype system, called BumbleBee. Finally, it evaluates the efficacy of an in-network processing facility with three distributed, connected applications. The chapter concludes with some questions that have not been answered.

5.1 Background

Applications are increasingly written in a containerized framework [15, 98, 129] as a collection of communicating microservices [102]. These frameworks provide many advantages: a strict decomposition of tasks, a consistent deployment model allowing in-place updates, declarative capture and preservation of system dependencies, and lightweight resource isolation and monitoring. Such ecosystems explicitly provide for

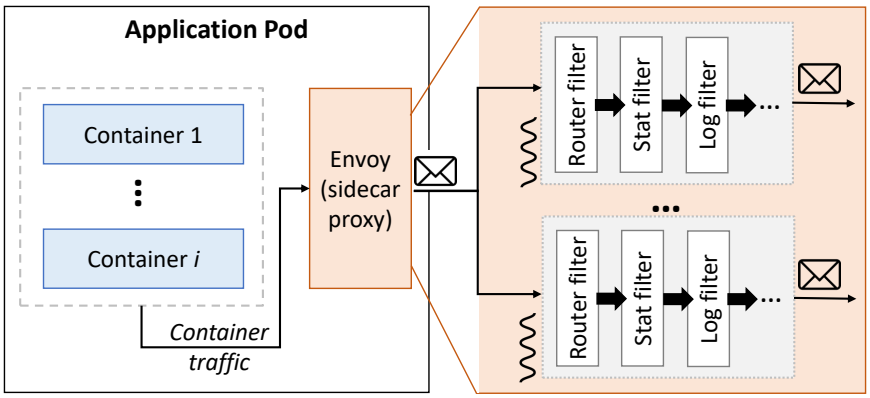


Figure 5.1: Envoy sidecars interpose on a pod's network communication.

separation of concerns through architectural decisions. Application writers need not be concerned with task creation, monitoring, placement, or scaling, relying instead on container orchestration frameworks [61, 143]. Likewise, they need not actively manage the communication between emplaced tasks. Instead, a service mesh [86, 128] provides reliable, fault-tolerant, load-balanced communication across complex topologies of task deployment. This section describes these frameworks, with an eye to BumbleBee's integration with them.

Containers: Docker [98] is a container-based virtualization platform that provides process-level performance and security isolation; such platforms have become the standard unit to manage and deploy software in the cloud. Container images include all of the user-level state required to launch an application, including binaries, support libraries, and configuration. Each container typically implements a single component microservice of the overall application, providing an API to the other constituent components.

Container Orchestration: Kubernetes [61] automates deployment, scaling, and management of distributed, containerized applications. The unit of deployment in Kubernetes is a *pod*. A pod is a set of containers that run under the same operating system kernel and share the same underlying physical resources, such as cores and disks. Because containers within a pod share a machine they can communicate cheaply

via local storage or intra-kernel messaging.

Developers write configuration manifests describing how Kubernetes should deploy an application on a set of physical or virtual machines, e.g., which container images to use, how containers are grouped into pods, and which ports each pod needs. The manifest also describes runtime goals for an application, such as pod replication factors, load balancing among replicas, and an auto-scaling policy.

Service Mesh: Service meshes [86] manage inter-pod communications within Kubernetes. They provide service discovery, peer health monitoring, routing, load balancing, authentication, and authorization. This is done via the *sidecar pattern* [18], in which a user-level network proxy called Envoy [76] is transparently interposed between each pod and its connection to the rest of the system; applications are oblivious to the sidecar and its mechanisms. Each Envoy instance is populated with `iptable` rules to route incoming and outgoing packets through the sidecar, as shown in Figure 5.1. This architecture makes the Envoy sidecar an ideal place to implement application-aware adaptation. It allows application writers to focus only on the needs of adaptation as data traverses the network, without having to integrate it with the application's behavior as prior systems did [106, 42]. We use the Istio [128] implementation in our prototype.

An Envoy sidecar has a pool of worker threads, mapped to the underlying threads exposed to this container. Workers block on ingress/egress sockets and are invoked on a per-message basis. On invocation, the Envoy worker passes the message through one or more application-specific *filters*. Filters are small, stateless code snippets that operate on individual messages. Filters have full access to a message and can perform simple operations, such as redirection, dropping, and payload transformation. Developers commonly use Envoy filters for monitoring and traffic shaping, such as collecting telemetry, load-balancing, and performing A/B testing. Envoy supports filters at several layers of the network stack.

5.2 Design Principles

To enable applications to adapt to a turbulent network environment systematically and efficiently, we incorporate design principles with the following principles in mind.

5.2.1 Problem: Separating adaptation logic from applications

Many applications today embed adaptation logic in their source codes to deal with a turbulent network environment. Video streaming applications, for instance, continuously monitor network conditions, and they switch video resolutions when throughput estimation changes. Sadly, applications receive little or no help at all from the underlying systems, because existing systems are oblivious to application fidelity. As a result, applications are responsible for two additional heavy duties—resource monitoring and taking adaptation actions—to adapt in turbulent network environments.

Unfortunately, the application-layer is ill-suited for monitoring resources, because they often rely on passive measurements. The video streaming applications estimate the current throughput based on the recent history of transmission time and size of chunks. So, when small, low-resolution chunks have been sent previously, they may underestimate the current throughput even if network conditions have improved. Further, the underlying system may increase or decrease shares of allocated resources to the applications even if there are no changes in available resources. Therefore, monitoring resources at the system-layer is not only more accurate but also removes burden from application developers.

Systems informing changes in resources to applications are useful, but are not good enough for applications to conquer volatile network conditions. Recall that adaptation logic elements are embedded in application source code. Consequently, changing the adaptation logic elements requires modifying the application itself and can be burdensome. Also, the logic cannot be easily ported to other applications to re-use.

Hence, separating adaptation layer from the core application functionality is critical. We find a sidecar local proxy is a good place to accommodate an execution of adaptation logic elements, and to separate them from the core application's business.

5.2.2 Problem: Generic abstraction for classes of applications

A sidecar proxy is a transparent user-level local proxy that can interpose an application's ingress and egress network traffic. The proxy may apply adaptation logics to a queue of in-flight data. The problem is that different types of applications also have different network transmission granularity, e.g., frames for vision and streaming applications, and sensor streams for Internet of Things applications. So, having a generic abstraction for classes of applications to perform adaptation is critical.

Luckily, applications relying on the connectivity have the common denominator of message granularity. Network protocols (e.g., HTTP and TCP) have request and response patterns, regardless of types of applications. That means that different applications can share a *message queue* abstraction. The abstraction is generic to applications but also to a range of operations such as reorder, drop, reroute, and defer messages. Furthermore, the crisp queue abstraction gives useful secondary metrics like age and latency for each message. Therefore, a crisp queue abstraction can support an array of adaptation logics without modifying applications.

5.2.3 Problem: Programming model for in-network scripting

A declarative programming model such as SQL can make it easy to implement static adaptation strategies, e.g., drop the frame if bandwidth is lower than 10 Mbps; it also requires declaring all adaptation points and subsequent actions to cover every possible corner cases in real-world deployment. So, declarative programming language is inadequate to flexibly deal with a turbulent network environment. On the other hand, imperative programming models such as Bash and Lua scripts are flexible and can provide a rich set of basic operations. With the message queue abstraction, the

run-time script can drop, reroute, and re-order in-flight messages. Further, the script can select actions by accessing the run-time information of each message (e.g., latency, age, available bandwidth). Therefore, the declarative programming model is a more adequate approach to express adaptation strategies.

Recall that the adaptation is done *in-network*, a critical path for connectivity. So, the script must be lightweight. Also, it must be simple to avoid potential errors arising from complicated logics; for complex operations, the script can utilize an asynchronous callback to collaborate with the application: the callback can inform the application about changes in network condition, and let the application adjust the resolution and fidelity of future messages. The callback can also invoke external services that transform in-queue messages. Lightweight in-network scripts with flexible asynchronous callback mechanism allow developers to control in-flight messages to adapt.

5.3 Design and Implementation of BumbleBee

This section discusses design and implementation details of a prototype, BumbleBee, that incorporates the design principles discussed in Section 5.2.

5.3.1 Queue manager

Recall that BumbleBee's core abstraction is a shared message queue; Envoy filters are stateless and created per message. As a result, BumbleBee extends Envoy with a separate queue-manager that runs concurrently with the worker threads. Worker threads pass messages to the queue-manager through the BumbleBee filter. The BumbleBee filter buffers data until a complete HTTP request or response that has been assembled and then forwards the message to the queue manager. For workloads that use other protocols besides HTTP, the BumbleBee TCP filter decodes messages in-situ based on specification of protocols.

The queue manager manages pairs of message queues (one for requests and one for

responses) and threads (one for each queue) and exposes a small number of methods that can be invoked by BumbleBee filter instances within worker threads. The queue manager creates a default pair of queues and threads, but applications can instruct the manager to create a pair of queues and threads to handle communication for pods with specific names. For example, in its orchestration configuration an application may name pods containing an object-detector running on the cloud "cloud-object-detector.local." It can then instruct BumbleBee to create queues in each pod's Envoy for handling requests to those pods. When a BumbleBee filter passes a message to the queue manager, the manager adds the message to the queue created to handle the message's destination (for outgoing messages) or source (for incoming messages).

The queue manager's thread pairs are primarily timer handlers that implement a token-bucket algorithm. On wakeup, a queue-manager thread checks its bucket to determine whether it has enough tokens to forward the head of its queue via Envoy's event dispatcher. The queue manager avoids spurious wakeups in two ways. First, when a queue is empty, its thread will block on a condition variable instead of polling. After a worker thread adds a new message to a queue, it signals the queue's condition variable. Second, when a queue is not empty, its thread sets its timer to wakeup when it will have enough tokens to forward the head of the queue. In our current implementation, the minimum sleep time is 1 ms.

5.3.2 In-network scripting

The BumbleBee filter and queue manager push and pop from shared queues, but BumbleBee applications need more ways to interact with messages. Fortunately, Envoy supports Lua scripting within filters, and BumbleBee leverages this feature to support application-aware adaptation. Similar to how a BumbleBee application creates custom message queues through its orchestration configuration, an application can also associate short Lua scripts with those queues. When a worker thread loads a BumbleBee filter, the filter reads the appropriate script from the orchestration

configuration and launches it within a Lua runtime.

We also considered declarative interfaces like YAML or SQL for in-network adaptation. However, we felt that an imperative approach would be easier for many developers, and Envoy's existing support for Lua made it easier to build a prototype. It is worth noting that Envoy recently announced support for running WebAssembly in filters, which could potentially allow BumbleBee to support any high-level language that compiles into WebAssembly bytecodes. We leave this for future work.

BumbleBee executes scripts *when messages arrive* for a purely practical reason: Envoy already supports running Lua scripts in filters. The primary drawback of running scripts when messages arrive is that it ties adaptation frequency to message arrival rate. In the worst case when messages stop arriving, an application's adaptation scripts will also stop executing. This design decision may be worth revisiting, but in our limited experience with BumbleBee, linking adaptation frequency to arrival rate has not caused any problems.

The main abstraction exported to BumbleBee scripts is a message queue. Each script can iterate queues maintained by the queue manager. To give scripts access to an underlying queue, BumbleBee uses Envoy macros to export a handful of C++ methods from the queue manager to the Lua runtime. These methods allow a script to access queue properties, such as its length, as well as apply coarse transformations such as dropping all messages, dropping the head of the queue, and reversing the queue's order. In addition to accessing a message queue, BumbleBee scripts can access queue-iterator and queue-item objects for traversing the queue and reasoning about individual messages, respectively. We use a simple, per-queue locking scheme to synchronize access to all queue-related states.

To help scripts make good adaptation decisions, BumbleBee exports a number of runtime performance metrics. At the lowest level, BumbleBee exposes TCP metrics, such as the congestion window size, number of in-flight packets, and mean round trip

time (RTT). BumbleBee also exposes the average end-to-end latency for messages in a queue, which Envoy calculates using request and response arrival times. In addition, BumbleBee provides information about how long each messages has spent in a queue through an object-item's age property.

Finally, network bandwidth is a crucial metric for numerous adaptation strategies. BumbleBee does not give script information about the available bandwidth along a physical link, but it calculates the observed bandwidth for messages forwarded from a particular queue. This allows scripts to reason about the observed bandwidth along their path of interest. For example, scripts can detect that a path has been disconnected if its observed bandwidth drops to zero.

5.3.3 External callbacks

BumbleBee's scripting environment allows applications to perform simple, local processing on enqueued messages. However, many applications can benefit from richer interactions between scripts and the rest of the application. For example, an application may want to use a script as an in-network watchpoint for changing conditions, such as notifying a pod when the observed bandwidth falls below a threshold. This is useful for real-time video-analytics applications that reduce the resolution of video frames sent to an inference model when bandwidth drops. Or an application may want to transform message payloads in ways that are too complex for a lightweight Lua runtime. This is useful for live video-streaming servers that want to downsample video chunks to prevent a hands-of-waving delay. To support this kind of functionality, BumbleBee allows scripts to make two asynchronous callbacks to external endpoints on Envoy's async HTTP client thread.

The simpler of the two is `notify (metrics)`. An application's orchestration configuration can bind a list of RESTful endpoints to `notify` invocations within a script. `notify` takes a string as an argument, and when a script calls `notify` the Lua runtime generates asynchronous HTTP calls with the string argument to any endpoints listed

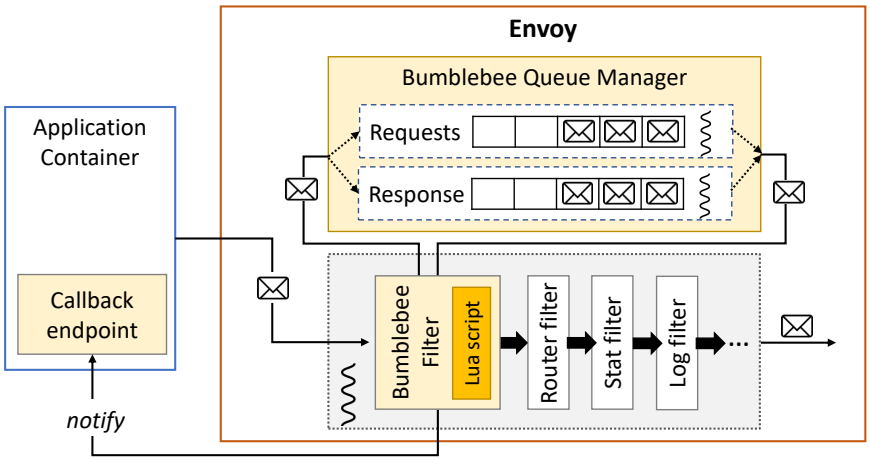


Figure 5.2: BumbleBee's integration with Envoy

in the orchestration configuration.

`transform()` is a more complex callback. As with `notify`, applications bind invocations of `transform` to a RESTful endpoint through their orchestration configuration. When a script calls `transform` on a queue entry, the Lua runtime marks the entry as "in progress" and asynchronously forwards the message payload to the registered endpoint. If a message moves to the front of the queue while marked in-progress, the queue-manager thread skips it, and sends the first entry not marked in-progress. When the endpoint returns with a transformed message payload, BumbleBee swaps in the new payload, updates the message header to reflect its new size, and remove its in-progress mark.

In our experience, `transform` is best used for complex computations that require specialized hardware, such as media transcoding. However, we believe that `transform` could also prove useful in cases where a transformation requires access to a state that is inaccessible to the BumbleBee filter, such as an external database. We are actively exploring such use cases.

5.3.4 Implementation of BumbleBee

BumbleBee is implemented in Envoy's HTTP and network filters. BumbleBee's HTTP filter currently supports HTTP/1.0 and Netty protocols. Other protocols such

Context	Interface	Description	Returns
Queue	length()	returns number of messages in a queue, useful to approximate queuing delay.	queue length
	avgLatency()	returns weighted moving average of end-to-end latency of messages (delta between request & response)	average latency
	observedBW()	returns observed bandwidth allocated to the queue–the rate of the queue sending data.	observed bandwidth
	TCPMetrics(m)	retrieves the TCP metrics (e.g., mean RTT) at the queue level.	TCP metric
	messages()	for-loop entry to iterate over messages in the queue.	message object
Message	size()	returns the size of the message's current payload.	size of payload
	age()	get the age, i.e., how long the message has been in the queue, in ms resolution.	age of message
	TCPMetrics(m)	retrieves the TCP metrics (e.g., mean RTT) at the message/request level.	TCP metric
	dst()	returns the current destination of the message.	message destination
	header()	returns the message's header.	message header
	bytes(i, j)	returns data from `i` to `j` of payload of the current message in raw binary format.	raw payload
	redirect(dst)	redirect the message to a new destination (dst).	
	transform(args)	asynchronously transform a message's payload by forwarding to a registered endpoint.	
	drop()	drops the current message from the queue. The function does not guarantee successful operation (e.g., already transmitted in the middle of dropping). If successful, returns the updated queue length, otherwise, returns the old queue length.	new queue length
	insert(msg)	inserts a new message `msg` after the current message in the queue. If successful, returns the updated queue length, otherwise, returns the old queue length.	new queue length
	moveToFront()	move the message to front of the queue.	
	moveToBack()	move the message to end of the queue.	
Callback	notify(metrics)	asynchronously send registered endpoints a metrics string.	

Table 5.1: BumbleBee interface for in-network scripting.

as HTTP/1.1 (WebSocket) and HTTPS can be supported without much engineering effort; we find many potential applications that can substantially benefit from BumbleBee can be configured to transport over either HTTP/1.0 or Netty protocols.

BumbleBee supports the in-network scripting through Lua programming language. Note that BumbleBee supports one Lua execution at a time; for long-lived tasks such as asynchronous callbacks, BumbleBee saves the current Lua runtime stack, and re-schedules when notified. BumbleBee supports only container-based platforms such as Kubernetes [61], because of the platform's growing popularity in real-world deployment; the fundamental design principles of BumbleBee are also applicable and extensible to also VM-based applications.

Metrics exposed: BumbleBee exports a number of network performance metrics on which to base adaptation decisions; these are summarized in Table 5.1. At the lowest level, BumbleBee exposes TCP metrics such as the congestion window size, number

of in-flight packets, and round-trip time (RTT). BumbleBee also exposes the average end-to-end latency for messages in a queue, which Envoy calculates using request and response arrival times. In addition, BumbleBee provides information about how long each message has spent in a queue through an object-item's age property.

Network bandwidth is a crucial metric for numerous adaptation strategies. BumbleBee does not measure available bandwidth along a physical link, but it calculates the observed bandwidth for messages forwarded from a particular queue. This allows scripts to reason about the observed bandwidth along their path of interest. For example, scripts can detect that a path has been disconnected if its observed bandwidth drops to zero.

5.4 Evaluation

To evaluate BumbleBee, we seek answers to the following questions:

- Does BumbleBee enable beneficial adaptation strategies?
- How difficult is writing adaptation strategies in BumbleBee?

To answer the first two questions we use our BumbleBee prototype to investigate adaptation strategies for three case-study applications. First, we use BumbleBee to help a distributed, vehicular-traffic monitoring application that adapts the quality of its object detection to changing network conditions. Second, we use BumbleBee to help a stream-processing application intelligently shed requests under bursty workloads. Finally, we use BumbleBee to help a live video-streaming service to reduce stalled playback while maintaining acceptable video resolution.

We run these workloads with BumbleBee using Istio 1.4.3, Envoy 1.13.0, and clusters of virtual machines managed by Azure Kubernetes Service (AKS) 1.18.14.

5.4.1 Case-study: traffic monitoring

Our first case study is a smart-city application that streams roadside video to machine-learning (ML) models. The ML models forward any detected vehicles to one or more traffic-light controllers. For each vehicle found, the models output a bounding box containing the vehicle and a confidence level. The application removes any bounding boxes with a confidence level below a threshold (e.g., 50%) and returns the filtered results to a traffic-controller. The controller uses the vehicle counts and locations to monitor and schedule traffic, such as reducing the time between green and red lights when road congestion is high.

Traffic monitoring is representative of many edge-computing applications [107]. The input sensors (e.g., roadside cameras) and controllers (e.g., traffic controllers) are co-located on the edge with a distributed computing pipeline between them. This pipeline must process sensor data fast enough for the controllers to respond to changes in the physical environment, and the application must operate even when network conditions are poor.

The ML pipeline can be instantiated along two paths: embedded in a resource-rich cloud environment or a lightweight edge environment. The cloud offers powerful machines and can support sophisticated and accurate ML models, whereas the edge can run a limited number of less accurate models. The application prefers results from cloud models, and it will send frames to the cloud as long as network conditions allow it.

Detection accuracy is a key measure of fidelity for traffic monitoring. Accuracy is highest when the network allows the application to stream high-resolution frames to the cloud, but as network conditions change, the application can adapt the video stream's quality by sending lower-resolution frames or reducing the frame rate. The application runs at lowest fidelity when it is disconnected from the cloud. During disconnections, the application must redirect video frames to its lightweight edge

models, sacrificing accuracy for availability.

```
function envoy_on_request(h)
  for queue in h:Queues():getQueue() do
    route = queue:route()
    if string.find(route, "cloud") then
      bw = queue:getBW()
      if bw == 0 then
        h:redirect("edge-detector")
      elseif bw < required then
        h:transform("180p")
      end
      if bw < required/2 then
        h:notify(bw)
      end
    end end end
```

Figure 5.3: This simple Lua script for the traffic-monitoring application redirects requests to the edge when the network becomes disconnected, down-samples enqueued requests when bandwidth drops, and invokes a registered callback when network conditions change significantly.

With BumbleBee, the application can implement these trade-offs using the simple Lua script in Figure 5.3. The script iterates over an egress request queue looking for entries destined for a cloud object-detector. When bandwidth drops to zero, BumbleBee redirects requests to the edge object-detector. If bandwidth falls below a threshold, BumbleBee forwards requests to the application's transform service, which reduces frames' resolution to 180p (320x180). And if bandwidth falls well below what is required, BumbleBee notifies the sender so that it can start to send lower-resolution frames.

Major cloud providers like AWS [9, 2], Azure [99, 10], and Google Cloud [48, 49] all suffer significant outages, and recent studies show that network conditions between the edge and cloud can be turbulent [107, 23, 150]. To understand how our traffic-monitoring application behaves when edge-to-cloud connectivity is poor, we run experiments with disconnections and constricted bandwidth between the edge pods and cloud pods. Note that for our experiments we logically divide cluster nodes between the edge and cloud, but the underlying physical machines and network are

entirely in Azure. We simulate a roadside camera by streaming a highway-traffic recording from Bangkok, Thailand [19]. We use YOLOv3 as our cloud object-detection model and TinyYOLO as our edge model. Both models are trained with the COCO dataset [89], which is designed to detect vehicles and passengers.

To evaluate whether the application benefits from BumbleBee, we measure the number of detected vehicles and end-to-end detection latency. The former metric influences how well the application controls traffic, and the latter influences how quickly the light controller responds to traffic changes.

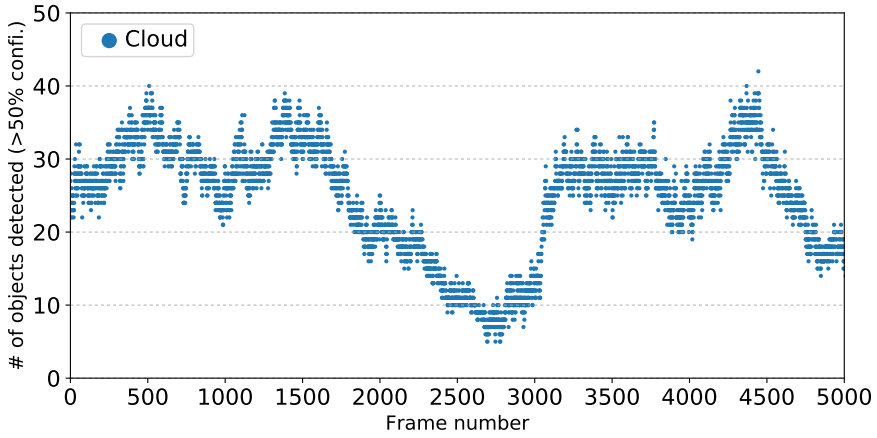


Figure 5.4: Our traffic-monitoring application performs best when fully connected to a YOLOv3 model in the cloud. This operating mode provides a baseline object-detection accuracy.

To characterize our traffic-monitoring application without BumbleBee, we first capture the baseline object-detection accuracy of streaming 360p (640x360) video at 15 fps when fully connected to the cloud. Figure 5.4 shows the number of detected vehicles over time with a confidence threshold above 50%. The YOLOv3 model in the cloud consistently detects between 10 and 40 vehicles.

To simulate a disconnected edge site, we run the application under BumbleBee and partition the edge and cloud pods after 1000 and 3000 frames so that the cloud object-detector is unreachable. We heal the network between frames 2000 and 3000. Loading tensor-flow models can be slow, so BumbleBee pre-loads the edge detector at

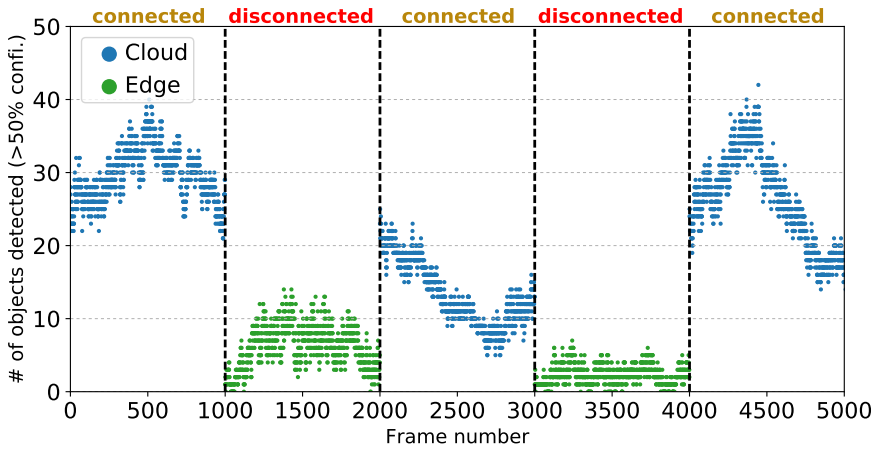


Figure 5.5: The application adapts when the edge becomes disconnected from the cloud. Instead of performing object detection with YOLOv3 in the cloud, BumbleBee redirects requests to a lightweight TinyYOLO object detector running on the edge. When the network heals, BumbleBee routes requests back to the cloud.

the beginning of the experiment. Figure 5.5 shows that the detected cars drop during disconnection, because the application switches to TinyYOLO on the edge.

There is a delay between when a disconnection occurs and when BumbleBee detects the disconnection. In our experiment, five frames stall before BumbleBee detects that the bandwidth is zero. Recall that our video streams at 15 fps, and so requests arrive every 67 ms. Thus, the first request sent after the disconnection experiences an approximately 350 ms of additional delay before BumbleBee redirects it to the edge. This is because four cloud-bound requests arrive after the first post-disconnection request but before BumbleBee detects the disconnection. When the sixth post-disconnection request arrives, BumbleBee has detected the disconnection and responds by redirecting all cloud-bound requests to the edge. Between disconnections, the application matches baseline detection accuracy. These results show that with BumbleBee, the application can continue to operate, albeit in a degraded mode, when the cloud is unavailable.

We also want to understand whether the application benefits from adapting to network changes that are less dramatic than a disconnection. Recall that end-to-end latency is a critical application metric. When disconnected, the weaker edge

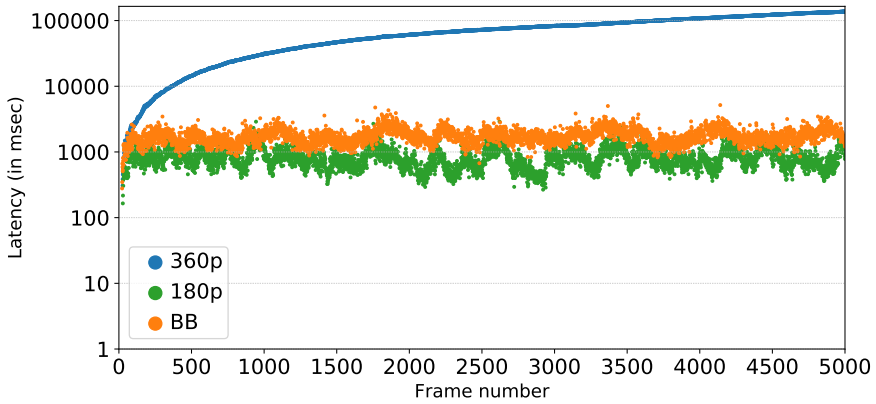


Figure 5.6: When edge-to-cloud bandwidth is 15 Mbps, sending 360p frames leads to head-of-line blocking and exponentially increasing detection latency. Sending 180p frames reduces median latency to 815 ms with no head-of-line blocking. BumbleBee (BB) allows the application to selectively downsample frames to balance latency (median latency 1700ms) and detection accuracy.

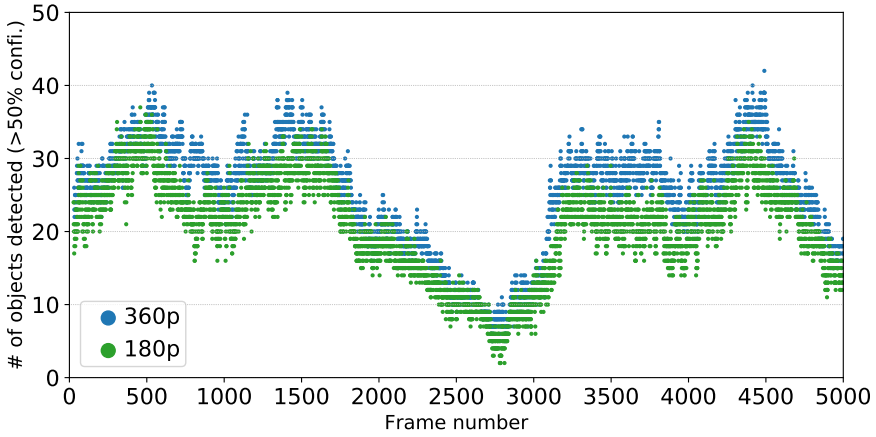


Figure 5.7: The cloud object detector identifies more vehicles with confidence greater than 50% in 360p frames than in 180p frames.

detector processes 360p frames 33% faster than the cloud detector using equivalent hardware. However, when bandwidth to the cloud drops, sending 360p frames can cause exponentially increasing queuing delay. To demonstrate, we restrict edge-to-cloud bandwidth to 15 Mbps and repeat the traffic-monitoring experiment twice, first sending 360p frames and second sending 180p frames. Frames are full-color, JPEG-compressed images. Figure 5.6 shows the results. When the application streams 360p frames (the blue line), the latency rises exponentially, but the median latency of streaming 180p frames is 772 ms.

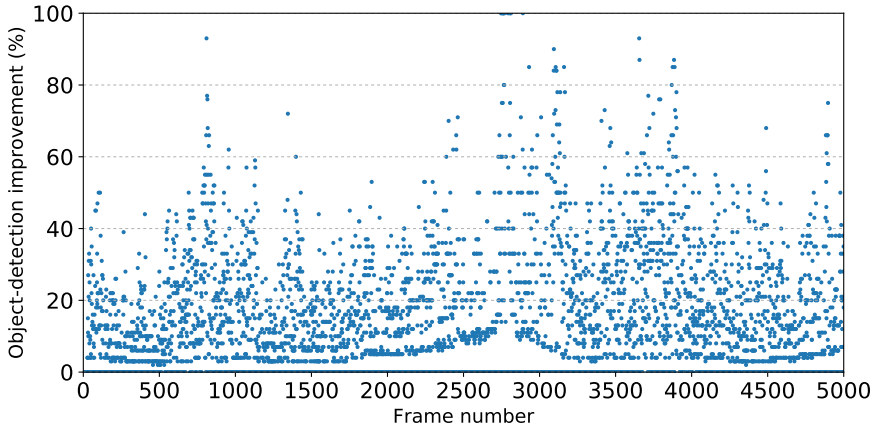


Figure 5.8: BumbleBee enables the traffic-monitoring application to send 360p frames when possible and avoid head-of-line-blocking by selectively downsampling frames to 180p. Each blue data point represents the percentage of additional objects that the BumbleBee-enabled application detects in a frame compared to sending all 180p frames. The 3444 frames (out of 5000) are downsampled to avoid exponential queuing delays. These frames gain zero percent improvement.

However, lower resolution frames reduce detection accuracy. Figure 5.7 shows the number of objects the application detects with confidence greater than 50% for 360p and 180p frames. Note that the blue dots are identical to those in Figure 5.4: 360p frames allow the cloud model to consistently detect more objects than the 180p stream, often significantly so. These results suggest that the traffic-monitoring application could benefit from selective adaptation by downsampling frames that cause queuing delay, and transmitting the remaining frames intact.

To confirm our hypothesis, we repeat our limited-bandwidth experiment using BumbleBee. The application sends 360p frames, and BumbleBee selectively downsamples frames that cause queuing delays. Figure 5.8 shows the percent improvement of the object detector with BumbleBee's selective downsampling enabled compared to always sending 180p frames. When BumbleBee downsamples a frame to 180p, the improvement percentage is zero. Overall, BumbleBee downsamples 3444 frames and leaves 1556 intact. Furthermore, the graph shows that selectively downsampling provides much better detection accuracy than always downsampling. Combined with the median latency of 1700ms in Figure 5.6, these results show that BumbleBee allows

the traffic-monitoring application to find a good balance between detection accuracy and latency using the simple script in Figure 5.3.

To summarize, the results show that our traffic-monitoring application benefits from BumbleBee in two ways. First, the application operates when disconnected from the cloud by redirecting requests to a weaker edge object-detector. Second, when network bandwidth constricts, the application selectively downsamples frames to balance end-to-end latency and detection accuracy. We also show that the adaptation strategies responsible for these benefits can be concisely expressed by the script in Figure 5.3.

5.4.2 Case-study: video streaming

For our second case study, we evaluate an HTTP Live Streaming (HLS) service with an Nginx server and HLS.js client [64]. At runtime, the server partitions an input live stream into a rolling sequence of self-descriptive, fixed-length MPEG-TS chunks at several resolutions. When a chunk can be downloaded, the server updates an HLS manifest file to announce its availability and resolution. The HLS client is responsible for all adaptation logic and periodically polls the manifest to learn when the newest chunk is ready. After reading the manifest, the client predicts the time to download the next chunk at the available resolutions. These predictions are based on the chunks' sizes and a bandwidth estimate calculated over a sliding window of prior downloads.

The client's competing objectives are continuous video playback and high video quality. Stalling occurs when the client's playback buffer is empty, which is far worse for the user experience than temporary drops in video quality [35]. For example, if bandwidth drops in the middle of downloading a chunk, the client's playback buffer may drain before the transfer completes. This is common when clients react too slowly to abrupt bandwidth drops and high variability [150, 95].

Prior solutions to this problem rely on either server [95] or client [150] modifica-

tions. Modifying a server without the clean separation provided by BumbleBee requires either building from scratch or understanding an existing codebase and continuously merging with external updates. Furthermore, client agnosticism is critical for open protocols like HLS, because service providers cannot dictate which of the many players a client may use [138]. Fortunately, BumbleBee can transparently apply a variety of adaptation strategies to correct an HLS client's bandwidth mis-predictions.

The BumbleBee script in Figure 5.9 illustrates such a strategy. The script adapts to sudden bandwidth changes faster than an unmodified HLS.js client by only considering the most recent chunk transfer rather than a sliding window over several transfers. Based on this bandwidth estimate, the script chooses among available chunk resolutions. If the requested resolution could cause a stall (line 6-9), BumbleBee modifies the path field of the HTTP header so that it refers to a lower-resolution chunk. However, if the client requests a resolution that could under-utilize the available bandwidth (line 10), BumbleBee swaps in a higher-resolution chunk path. BumbleBee's bandwidth estimate requires Envoy modifications to monitor low-level transfer progress or a service provider to place a middlebox between the client and server. For the purposes of our experiments, we emulate the latter by co-locating a proxy with the client and configuring the client to direct its requests through the proxy. Future versions of BumbleBee will include the necessary Envoy modifications.

We first evaluate the video-streaming service with two synthetic bandwidth changes: a sudden drop and recovery and a gradual drop and recovery. These changes highlight the trade-offs of reacting more quickly than the HLS client's strategy. In addition, to evaluate the efficacy of BumbleBee in real-world scenarios, we analyze network-condition logs of Puffer [150] clients watching live video streams. We limit our experiments to traces that cause stalls of more than 100 s, and from these traces replay estimated instantaneous bandwidth conditions. We replay the first ten minutes of each trace.

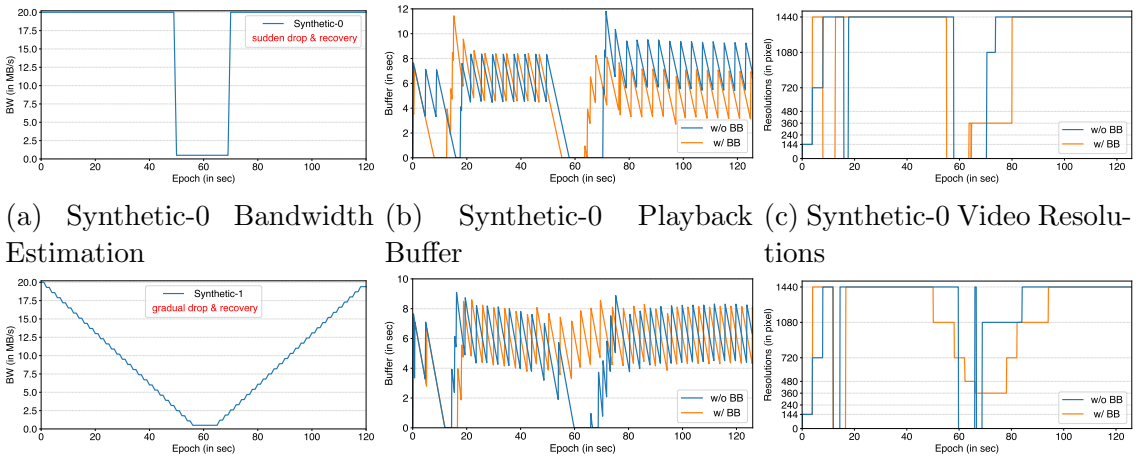
```
function envoy_on_request(h)
  hdr = h:header()
  bw = hdr:get("bw-est")
  curr, chunk = hdr:get("path")
  -- use bw estimate to choose a chunk
  pred = find_resolution(bw)
  if pred < curr then
    -- downsamples
    hdr:replace("path", pred.. "/".. chunk)
  elseif pred > curr * 2 then
    -- upsamples
    hdr:replace("path", pred.. "/".. chunk)
  end end
```

Figure 5.9: This BumbleBee script for the video streaming application predicts appropriate resolution to transmit based on the most recent bandwidth measurement and distribution of chunk sizes. When the script disagrees with the client, it overwrites the `path` of chunk's resolution to increase or decrease resolutions. Note that the script is conservative about upsampling to avoid potential stalls.

For all experiments we run the Nginx server under Kubernetes in a dedicated virtual machine with an Nvidia V100 GPU, 6 vCPUs, and 112 GB of RAM. For the client, the HLS.js player is in the same data center as the Kubernetes cluster but in a separate virtual machine with sufficient underlying bandwidth between the two. We use Linux `TC` to replay bandwidth traces at the client side, and we use the default player configuration unless noted. Each video chunk is four-seconds long.

We characterize streaming with and without BumbleBee with two metrics: playback buffer and video resolution. The playback buffer is the seconds of video that a client can play without receiving new data from the server. When the buffer reaches zero, the video stalls. Resolution represents video quality. Buffer and resolution can be traded off. In the extremes, sending only low-resolution chunks minimizes quality but maximizes buffer, and sending only high-resolution chunks maximizes quality but minimizes buffer. Because stalls are so painful [35], BumbleBee wants to offer acceptable quality with minimum stalls.

Figure 5.10a shows the first synthetic trace: a sharp bandwidth drop for 20 s and fast recovery. Figures 5.10b and 5.10c show the clients' playback buffer levels



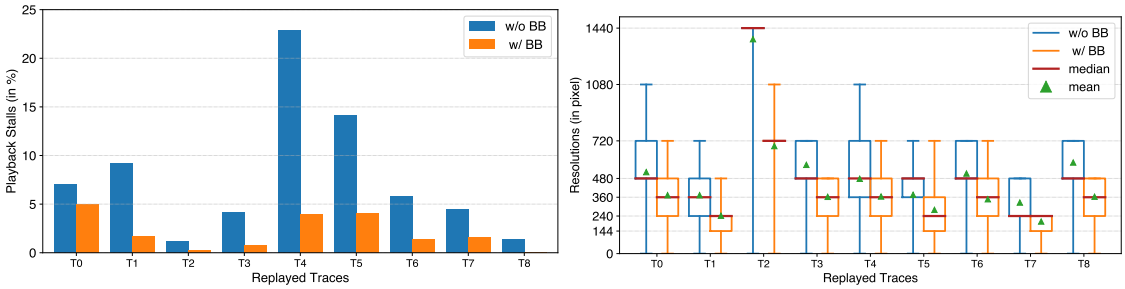
(a) Synthetic-0 Bandwidth Estimation (b) Synthetic-0 Playback Buffer (c) Synthetic-0 Video Resolutions

(d) Synthetic-1 Bandwidth Estimation (e) Synthetic-1 Playback Buffer (f) Synthetic-1 Video Resolutions

Figure 5.10: BumbleBee helps the live video streaming application to adapt quickly and cautiously. Figures in the first column show the bandwidth estimates for both traces. Figures in the second column show how the client's playback buffer changes during the trace. Figures in the last column demonstrate fast and agile adaptations by BumbleBee's script.

and displayed resolutions over the course of the trace, respectively. The client under both configurations stalls as it calibrates its bandwidth estimates. The client under both configurations also stalls when bandwidth drops. However, under BumbleBee the client adapts to the drop and rebuilds its playback buffer faster than without BumbleBee. Overall, BumbleBee reduces stalling from 13 s to 9 s, a 32% improvement. As Figure 5.10c shows this is possible because under BumbleBee the client reduces its resolution to 360p near 65 s, whereas without BumbleBee the client continues to download 1080p chunks.

Figure 5.10d shows the second synthetic trace: gradual bandwidth decrease and recovery, each over 50 s. We hypothesized that BumbleBee would offer little benefit on this trace, anticipating that the client's default estimates would closely track the gradual changes. Surprisingly, Figure 5.10e shows that BumbleBee reduces post-calibration stalling from 11 s to 5 s, a 55% improvement. Figure 5.10f shows that without BumbleBee the client fails to adapt to decreasing bandwidth, continuing



(a) Across all traces, playback stalls are significantly less with BumbleBee than without.

(b) To eliminate stalls, BumbleBee sacrifices some video resolution.

Figure 5.11: Experiments with the nine Puffer traces with the most stalls show how BumbleBee helps the live video streaming application to reduce stalls while maintaining acceptable video resolution.

to fetch 1440p chunks. In comparison, BumbleBee reduces resolution in a step-wise fashion and eliminates all stalling in the valley.

We repeat the experiments with nine Puffer traces. Figure 5.11a summarizes the percentage of total stall time that a client experiences during each trace, with and without BumbleBee. The client with BumbleBee stalls at the most 5% of the total duration, and the client without BumbleBee stalls 22% of the time, a 77% improvement. Figure 5.11b shows box plots of playback resolution, including mean and median. Note that in trace T2, which exhibits the least stalling without BumbleBee, the HLS.js client achieves higher resolutions than with BumbleBee albeit with some additional stalling. From the logs, we find that BumbleBee's script is too cautious about sending higher resolutions that could clog the connection during T2. This matches our expectation that quickly reacting to network changes to aggressively avoid stalling can lead to worse bandwidth utilization.

5.4.3 Case-study: stream processing

Our final case study is the Yahoo! stream-processing benchmark [25] that counts ad views from an input stream of ad impressions, i.e., clicks, purchases, and views. The benchmark is widely used [141, 65, 94, 154], because it mimics in-production

workloads and business logic. The first stage reads and parses impression data, the second stage filters out non-view events, and the final stage stores aggregate view counts over 10 s sliding windows. Impression counts help ad services bill customers and select the next ads to display. In the latter case, *timeliness* (meeting a latency deadline) is more important than *completeness* (fully processing every input), and many practitioner testimonials [140, 39, 40] emphasize the importance of timeliness.

By default, the Yahoo! benchmark generates emulated impressions at a constant rate, but real-world rates can be bursty. Bursts may be problematic for applications with timeliness requirements, because practitioners often statically allocate resources and must restart pipelines to scale dynamically [141]. Over-provisioning is not always possible, and unexpected bursts can rapidly increase end-to-end latency as applications fall behind processing every message.

Load-shedding [136, 135, 149] is a common way to adapt to such bursts. Shedding trades completeness for timeliness by dropping less important inputs to free resources and improving the number of deadlines met. Today this adaptation strategy can only be implemented by modifying an application's internals, but BumbleBee can intelligently shed load for unmodified applications.

To characterize how effectively BumbleBee helps the Yahoo! benchmark improve timeliness, we orchestrate the benchmark with Kubernetes by placing a containerized Apache Flink [22] worker in a pod. A worker pod can execute any stage and can pass inter-stage data within the same pod. Each Kubernetes node hosts one pod and is a virtual machine with two vCPUs and 8 GB RAM, connected by an underlying network provisioned at 1 Gbps. The benchmark polls external Kafka brokers for input events and stores results in an external Redis database. 10 s sliding windows are too coarse to properly measure the impact of bursts on timeliness, so we add a small amount of instrumentation to aggregate over 1 s sliding windows.

Figure 5.12 shows a BumbleBee script that uses custom message-dropping logic

```
1  filt_thrd = 0.5 --- filtering threshold in sec
2  late_thrd = 1.0 --- lateness threshold in sec
3  function envoy_on_response(h)
4   queues = h:Queues()
5   for queue in queues:getQueue() do
6     for msg in queue:messages() do
7       json = msg:json()
8       if queue:avgLatency() > filt_thrd then
9         event_type = json:getString("event_type")
10        if event_type ~= "view" then
11          msg:drop()  --- preemptively filter
12        end
13      end
14
15      event_time = json:getNum("event_time")
16      age = h:epoch() - event_time
17      if age > late_thrd then
18        msg:drop()    --- drop late msgs
19      end
20    end end end
```

Figure 5.12: This BumbleBee script pre-emptively filters messages and drops late messages to save inter-pod bandwidth when it detects latency in the pipeline.

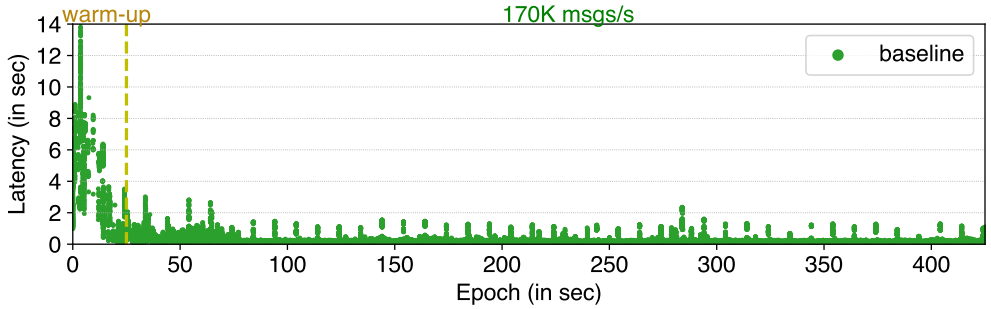


Figure 5.13: Our stream-processing application processes input messages mostly under 2 s latency after a short warm-up period, when 170k input messages are streamed per second.

to implement two forms of load shedding: preemptive filtering and dropping late messages. Recall that the benchmark filters out click and purchase events in its second stage. Under BumbleBee, if latency increases, the benchmark preemptively filters non-view events before the second stage (lines 8-13). The script also drops view events if they are unlikely to meet their deadline (lines 17-19). Both adaptations free resources as the script detects latency in the pipeline.

Our baseline benchmark configuration runs under Kubernetes, without an Envoy

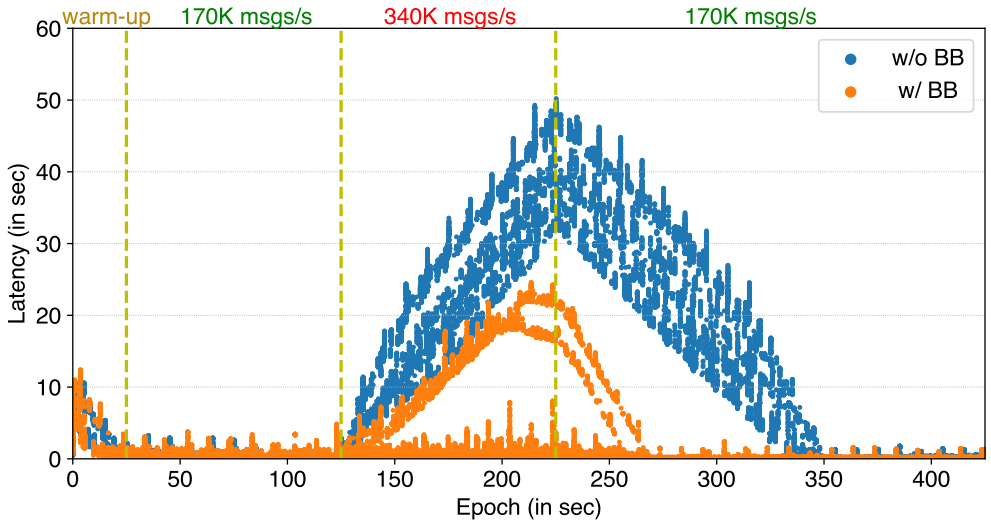


Figure 5.14: Temporal 2x input load spikes leads the application to experience high latency for 1.5 times longer than the spike duration even after the input load returns back to the previous level. The BumbleBee-enabled application takes less than 30 s to bring latency back to the previous level.

sidecar or BumbleBee. We first run the baseline configuration with a constant, baseline load of 170k events per second. To characterize latency, we sample the end-to-end latency of the last event included in the benchmark's 1 s aggregation window. Figure 5.13 shows how the latency of these sampled events change over time. Latency for the first 25 s is highly variable as the benchmark warms up. After the warmup, sampled latency is largely under 2 s. This is expected since we provisioned enough compute and network resources to process every event within 2 s.

We next run an experiment with variable load: first 170k events per second for 125 s, followed by a burst of 340k events per second for 100s, followed by a return to 170k events per second for 200s. Figure 5.14 shows sampled latency for the baseline benchmark (w/o BB) and the benchmark with BumbleBee (w/ BB) under variable load. During the burst, BumbleBee drops over 29% of all events, and after the burst, BumbleBee drops less than 6% of events. Compared to the baseline, BumbleBee's custom dropping policy significantly improves sampled latency and time to recovery. Excluding warmup, BumbleBee allows nearly 74% of sampled views to be processed

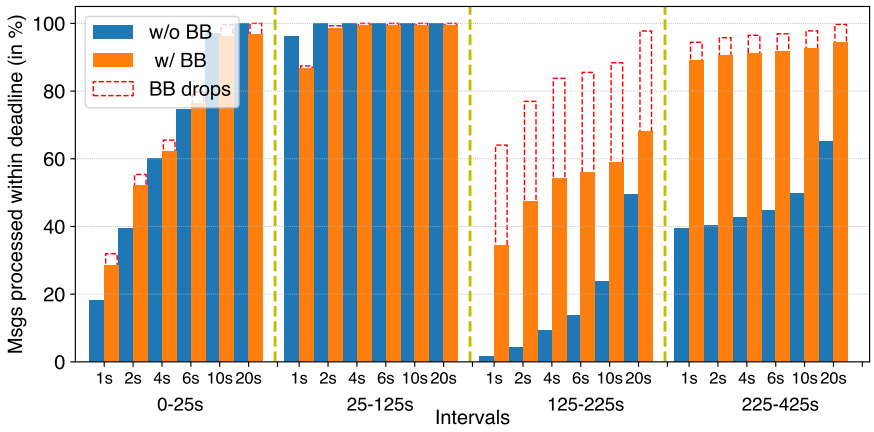


Figure 5.15: When the input load increases above expected level that operators have projected and have provisioned resources accordingly, the application hardly processes messages within a deadline. The consequence continues to stay longer than the ramp-up period.

within 2 s, whereas the baseline benchmark allows only 44%. In addition, BumbleBee returns the benchmark to steady state less than 50 s after the burst ends; without BumbleBee, it returns to steady state after 125 s.

A limitation of the current BumbleBee implementation causes the two arcs in Figure 5.14 that peak at 20 s and 25 s sampled latency. BumbleBee intercepts only inter-pod communication, but benchmark pods contain workers for all stages. Thus, sometimes the benchmark transfers data between stage workers residing in the same pod, i.e., over local Unix sockets on which BumbleBee cannot interpose. This phenomenon is an artifact of the Yahoo! benchmark's design and would not be an issue for applications that separate each stage into a dedicated tier of pods.

Figure 5.15 highlights how BumbleBee impacts meeting deadlines of 1-20 s during the warmup, baseline-load, bursty-load, and second-baseline intervals. As expected, longer deadlines (e.g., 20 s) are met more often than shorter ones (e.g., 1 s) with and without BumbleBee. There is also little difference between the two configurations during the initial warmup and baseline intervals. However, BumbleBee provides substantial benefit during the bursty interval, allowing the benchmark to meet nearly 23x more 1 s deadlines and 37.8% more 20 s deadlines than without BumbleBee. Bumble-

Bee provides substantial benefits when load returns to normal, allowing the benchmark to meet over 90% of its deadlines, regardless of length. In contrast, the baseline benchmark only meets less than 40% of its 1 s deadlines and 65% of its 20 s deadlines. This is due to the baseline benchmark's emphasis on completeness, and having to work through its backlog of enqueued events even after load has returned to normal.

5.5 Latency Micro-benchmarks

To evaluate the overhead imposed by BumbleBee compared to Envoy, we measure end-to-end latency using the HTTP benchmarking tool wrk2 [148]. The tool generates HTTP requests at a constant rate and outputs a latency distribution. We configure wrk2 to generate 500 requests per second with 1000 concurrent connections over five one-minute runs.

For our experiments, we create a client pod that runs wrk2 and assign it to a GPU node. We also create a server pod running the Nginx web server under default settings on a normal node. Both pods contain an Envoy sidecar with access to two cores. We measure the latency distribution under four client configurations: (1) Envoy without BumbleBee, (2) BumbleBee with no Lua script, (3) BumbleBee with a simple queue-iteration script, and (4) BumbleBee with a simple LIFO (Last In First Out) script. The first configuration serves as baselines for understanding BumbleBee's scripting overhead. Note that all configurations with BumbleBee move messages from the BumbleBee filter to the queue manager.

We use a simple LIFO and queue-iteration scripts used in the experiments. BumbleBee's queues are internally implemented as doubly-linked lists, which makes LIFO reordering relatively inexpensive. However, iterating over the queue could be slow for two reasons. First, BumbleBee uses a per-queue locking scheme that ensures only one script can execute at a time. Second, the Lua runtime creates a new stack and object bindings each time the iteration script runs. These startup costs are draw-

backs of using a scripting language instead of binary executables or bytecodes like WebAssembly.

To test our hypothesis, we run wrk2 five times with each client configuration. Up to the 75th percentile, the latency for all BumbleBee configurations is very close to Envoy, between 6.5% to 12% extra overhead, where the absolute value for the latency overhead is between $0.15ms$ and $0.35ms$. However, the cost of iterating over the queue is apparent at the very tail of the distribution. For example, at the 90th percentile, the iteration script's latency is 23% more than Envoy's, and at the 99th percentile it is 9.5% more. To further quantify BumbleBee's latency overhead when iterating the queue, we ran a micro-benchmark over a 10,000 length queue and called the `size()` function on each request. We measured the amortized latency overhead of iterating over a single request and found this value to be negligible, an average of $0.26\mu s$.

5.6 Related Works

Adaptation in mobile computing: Resources in mobile computing are highly constrained as opposed to a data-center environment. Prior adaptation systems [106, 42] trade application fidelity for various metrics. Similar to BumbleBee's callback functionality, Odyssey [106] create a collaborative adaptation solution that notifies applications to adapt their fidelity. On-demand distillation [42] performs "on-the-fly" adaptive transcoding of web contents based on the client's bandwidth, similar to BumbleBee's dynamic transformation. However, these systems do no expose control over the enqueued data which is critical to correcting "head-of-line" blocking issues.

Many others [12, 30, 26] integrate adaptation logic for better use of computation resources. Cyber foraging [12] is a runtime framework that allows developers to write and deploy complex adaptation tactics. MAUI [30] and CloneCloud [26] partition application code, either with developer-defined annotations (MAUI) or through static analysis (CloneCloud). Then, they adaptively offload partitions between local execu-

tion (on the mobile device) to remote execution. BumbleBee can be thought of as an extension to these systems where it can redirect the offloading traffic based on runtime variables such as network bandwidth.

Adaptation in video streaming: Video streaming [150, 119, 95, 67, 131, 71] is another domain that employs various adaptation strategies to improve video watching experience. A few recent works [150, 95] propose video streaming servers that adaptively select the best bit-rate by using machine learning to predict the bandwidth or transmission time. Others have developed video clients to adapt to network conditions changes for fairness [71] and stability [67], to minimize rebuffering [131], and to handle unexpected network outage [119]. While the individual solutions vary, these solutions can easily be reimplemented in BumbleBee and can leverage the low-level networking metrics and control available with BumbleBee to achieve improved performance (as shown in Section 5.4.2).

Other Adaptations: Odessa [120] is an adaptive runtime for partitioning and executing computer vision application remotely. The runtime balances the level of pipelining and data-parallelism to achieve low latency under variable network conditions. Kahawai [31] is a system for cloud gaming where clients with modest local GPUs collaborate with powerful cloud servers to generate high-fidelity frames. Kahawai adapts to network changes by adjusting the fidelity and frame rate of frames. Outatime [81] is a speculative execution system for cloud gaming where thin-clients send input, and servers at the cloud render speculative frames of future possible outcomes while adapting to network tail latencies. These systems can leverage the scripting interface and in-network processing capabilities of BumbleBee to improve or simplify their adaptation strategy.

In-network Processing: The concept of in-network processing was proposed over two decades ago for where custom in-network applications are deployed at the router to provide additional functionalities, e.g., webpage caching [137]. Recent develop-

ments in networking hardware (e.g., smart NIC, FPGA) have led to revisiting the idea of in-network processing. Flexible programming languages such as P4 [17] have emerged to simplify the development of in-network processing applications. As a result, many [84, 83, 52, 134] have explored using in-network processing for a wide variety of use cases such as improving consensus protocols (NOPaxos [84]), faster transactions (Eris [83]), network telemetry (Sonata [52]), or improving network functionalities, e.g., DNS and NAT (Emu [134]). Along the lines of these works, BumbleBee allows in-network processing of custom adaptation logic but for containerized environments such as Kubernetes.

5.7 Discussion

Despite the vastly improved network conditions, applications still encounter *uncertainty* that stems from turbulent network environments. Applications employ custom adaptation logic to deal with the uncertainty; their limited view in network leads to performance degradation. In this chapter, we describe BumbleBee, a set of extensions to support application adaptation *in network*.

BumbleBee is integrated with the container orchestration and service-mesh that supports container-based microservices. This is done by judiciously widening the in-network interface in two ways. From above, applications supply simple scripts that describe adaptive logic. From below, service mesh sidecars expose the queue of pending messages so that these scripts can drop, reorder, redirect, or transform those messages. Experiments with a BumbleBee prototype demonstrate the benefits of our approach: (1) by using BumbleBee, ML applications at the edge can utilize cloud resources when available and operate without interruption when disconnected, (2) BumbleBee increases the number of deadlines met by 23x more 1 s deadlines and 37.8% more 20 s deadlines on the Yahoo! stream-processing benchmark, and (3) BumbleBee reduces stalled playback by 77% during HLS video streaming under real-world

network conditions.

CHAPTER VI

Conclusion

6.1 Thesis Contributions

Application connectivity has a profound impact on society. Sadly, existing underlying systems are largely oblivious to turbulent network environments that applications suffer from. This dissertation presents evidence, problems, and solutions to support the following thesis statement: *Emerging distributed, connected application architecture suffers from turbulent network environments. Network-conscious systems for the applications are beneficial, effective, and valuable.* Specifically, this dissertation makes the following contributions.

Measurement Studies: To identify problems encountered by applications in vehicular setups and data-center environments, I conducted a set of measurement studies. The studies make contributions in three ways. First, I demonstrated that each network has a distinct RTT distribution and availability of network interfaces equipped in vehicles sold today. The tails of RTT distributions are extremely high. Next, I showed a potential latency improvement with cellular edge nodes. The measurement results show that latency to an associated node is an order of magnitude better than reaching the cloud; reaching a nearby but non-associated node can result in higher latency than reaching the cloud. Lastly, I demonstrated a turbulence in the data-center network environment. Bandwidth measurement results suggest that even well-provisioned data-center environments can be highly turbulent. The results of

measurement studies had motivated building of *network-conscious* systems to improve application connectivity in fundamentally turbulent network environments for distributed, connected application architecture.

Strategic Redundancy: The measurement study in vehicular setups clearly indicates that network interfaces in vehicles have distinct performance and availability distributions. It also shows that predicting near-future RTTs of individual wireless networks is extremely challenging. One network sometimes performs better than others but not always. When prediction is uncertain, redundantly transmitting data over multiple network interfaces and using whichever arrives the fastest can vastly improve application connectivity. Unfortunately, redundancy can be very wasteful, especially when the best performing network is known. Strategic redundancy balances resource usage and performance benefit by hedging a bet against uncertainty based on previous RTTs observed. The prototype MPTCP-kernel scheduler, called RAVEN, builds confidence intervals for each network to determine when to employ redundancy. The evaluation in live experiments shows up to 11x in 95% tail latency improvement, as compared to MPTCP's default scheduler. Strategic redundancy is clearly beneficial to improve application-perceiving latency.

Co-locating Computations: The measurement study in cellular edge nodes shows that associated nodes offer low latency; using a wrong node, i.e., not associated, can lead to latency much worsen than using the cloud. A state-of-the-art migration system supports pre-provisioning that moves the majority of a static state ahead of time. To enable pre-provisioning, we explored three next node prediction methods. None of the individual methods offered a good prediction accuracy, but a joint prediction method provided a better accuracy. Unfortunately, our preliminary evaluation with the existing system, highly optimized for cellular edge nodes, shows a high downtime for two interactive applications used in vehicles. This is acute for emerging vehicular applications that generate a large final delta to transfer and when inter-tower bandwidth is

prohibitively constrained. We identified that intermediate results and residuals from the application execution consists of a major portion of state transfer. To enable a fast migration, we introduced an Inter-request State (IRS) abstraction that collaborates with an application to ship necessary bytes for an internal application state. We designed and implemented a prototype module, called Croesus. The evaluation results of Croesus show up to two orders of magnitude improvement in cumulative downtime, compared to QEMU, while saving bandwidth up to six orders of magnitude. Thus, computation co-location is valuable to improve application connectivity with a fast migration.

In-network Application-aware Adaptation: Emerging hybrid application architecture that spans across edges and clouds can painfully suffer from turbulent network environments. The measurement studies in data-center networking illustrate volatile goodput observed in inter- and intra-DC connections. The problem is acute for hybrid application architecture that spans across not only edges and clouds, but also, geo-distributed clouds. When network conditions change, only applications know the best way to adjust their fidelity. Unfortunately, existing systems provide little-to-no support for in-network application-aware adaptation. We elaborate core design principles for a lightweight in-network facility, located in the middle of a critical network path. We implemented a prototype filter, called BumbleBee, in a sidecar proxy Envoy in popular containerized platform Kubernetes. BumbleBee allows application developers to inject adaptation scripts that execute per message requests. BumbleBee also exposes queue abstraction to scripts to modify payloads and/or orders of transmission. Experiments with a BumbleBee prototype show that edge ML-workloads can adapt to network variability and survive disconnections, edge stream-processing workloads can improve benchmark results by 23x, and HLS video-streaming can reduce stalled playback by 77%. Hence, in-network application-aware adaptation is effective at coping with turbulent network environments for emerging hybrid architectural applications.

6.2 Thesis Limitations

This dissertation has a few limitations that may lead to potential future works. First, mode-switching in the strategic redundancy is less effective for obvious bandwidth-heavy application such as video streaming, because RAVEN waits for network queues to fill up with redundant data before switching to striping mode. It could have used application information like the HTTP header to adjust mode-switching in the beginning of transmission, rather than waiting for queues to fill up. Next, RAVEN prototype does not consider the cellular modem's power-nap mode that puts the modem into idling when unused for a period. As a result, RTT predictions can be frequently inaccurate due to waking-up latency. RTT prediction mechanism in the prototype uses an offline trained model. This can be a significant drawback when new network environments are encountered. One alternative is to have an online training process in user-space that shares prediction with the kernel scheduler.

Secondly, the size of the application state can be a significant bottleneck in migration time for the Croesus prototype. Naive application developers can annotate and include unnecessary contents, resulting in a large state to transfer. A potential optimization includes offline profiling that provides a relationship between expected downtime and variables in IRS. Also, the Croesus prototype requires applications to maintain casual message ordering. A message's sequence is exposed to applications; in case of transient mobile networking, applications need to handle failed and delayed messages. The Croesus prototype can enforce strict message ordering through buffer-to-forward mechanism, but it incurs significant delays in migration completion time. This is a limitation from user-space implementation. The Croesus prototype operates at Python's run-time level, which supports only applications written in specific programming language.

Finally, the BumbleBee prototype has some technical limitations. The current version of BumbleBee supports only message-based protocols like HTTP/1 and

Netty [97]. Stream-based bi-directional protocols such as HTTP/2 and WebRTC have no clear boundary on per-data. Also, a packet of those protocols contains fractions of multiple messages. A future version of BumbleBee may work with those protocols; BumbleBee needs to parse and decipher messages, which can add significant overhead in the middle of a critical network path. Another implementation drawback that we are aware of is the single global lock on Lua script executions. BumbleBee does not support concurrent script executions. We have not found a compelling use-case to support concurrent script executions. However, emerging data-intensive applications like industrial sensor streaming applications may need concurrent execution of multiple scripts. The last limitation is that BumbleBee operates on a *per-node* basis, lacking a global view of shared network resources. This is similar to congestion control algorithms that try to obtain optimal congestion windows. This limitation can lead to an outcome without Nash equilibrium, e.g., under- and over-utilized bandwidth allocations by competing applications. Extended BumbleBee can benefit from having global network resource monitoring.

6.3 Future Work

While application connectivity may have not grown as much as Cisco predicted five years ago [68], its importance has become clear to our society. Advances in radio and hardware are fundamentally limited to further improve application connectivity. For instance, a packet traveling at the light of speed from New York to San Francisco takes 31 ms [1] in RTT. To overcome the physical constraints, Content Delivery Network (CDN) has been widely deployed by content providers such as YouTube and Netflix. They cache popular video streams in regional servers, located closer to end users, to provide lower latency, higher throughput (with less packet loss), and ultimately, better user-experience. The idea of emerging edge-computing follows a similar principle, but

[1] Road distance / speed of light = $4671km/299km/ms \approx 15.6ms$

mostly for computation.

A packet traversing from source to sink visits several to tens of network hops that can potentially provide a fraction of resources. For the remainder of this section, I outline potential future areas that can further improve application connectivity with potential in-network computing infrastructure.

Delay-conscious Fractional Computing: Links between hops incur transmission delays. The idea is that while waiting for data to be acknowledged, we can mark a fraction of data and do computation on it. A fundamental design challenge is to understand a clear boundary of mutually exclusive partition of data. This can be particularly challenging for domain-specific workloads such as a video frame. Object detection works on an entire single frame or even a slice of frame. The later partitioning can lead to inaccurate results.

A less important but crucial design aspect is the workload scheduler that accurately predicts a delay between links and partitions data to process accordingly. Latency is dictated by the last hop that finishes. A system can employ workload partition techniques that are thoroughly studied in the stream-processing community.

Differential Encapsulation Platform: Cloud- and edge-computing environments have only two encapsulation platforms: a full-fledged VM and a lightweight container. VM technology such as QEMU provides a full isolation through hardware ring protection and runs applications in guest OS. VM is assigned with a dedicated amount of resources when spawned. On the other hand, container technology such as Docker and containerd are much more lightweight because they run in host OS. Container provides weaker isolation via Linux cgroups. Further, it has no control over the resources allocated.

Clearly, each platform has pros and cons: strong isolation vs. less resource consumption, dynamic resource allocation vs. dedicated, etc. The problem is that application developers need to choose one of two platforms. One can imagine an application

that needs dynamic preemptive resource allocation with a strong isolation guarantee. Or, based on resource utilization and SLA, a service provider can adjust features provided to applications. Changes in in-network resources are frequent, so does features in-network applications prefer and need. Lack of differential encapsulation platforms can be practical drawback for deploying in-network computing environments.

6.4 The End: Summary

Application connectivity plays a critical role in today's society: sharing an opinion on social media, live video-streaming a historic moment, warning a driver about hazardous road conditions, etc. Unfortunately, application connectivity painfully suffers from *turbulent* network environments, because existing underlying systems are largely *oblivious* to network turbulence, resulting in poor application performance and worse user-experience altogether.

This dissertation lays basic building blocks toward *network-conscious* systems to tackle problems encountered by emerging distributed, connection applications. It first reports a series of measurement results to demonstrate the status quo that applications painfully suffer from turbulent network environments. Based on the lesson learned, it introduces three prototype systems that incorporate a set of design principles: strategic redundancy, state segregation, and in-network application-aware adaptation. Strategic redundancy in a high-mobility setup allows hedging a bet against uncertainty in future connectivity performance. State segregation enables co-locating computation closer to high-mobility devices with fast migration time. In-network application-aware adaptation allows developers to inject lightweight scripts to quickly react to changes in network conditions. Finally, it concludes with potential research directions that can further improve application connectivity.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] M Allman, K Avrachenkov, U Ayesta, J Blanton, and P Hurtig. Early retransmit for TCP and stream control transmission protocol (SCTP). IETF RFC 5827, 2010.

[2] Amazon's massive AWS outage was caused by human error. https://www.vox.com/2017/3/2/14792636/amazon-aws-internet-outage-cause-human-error-incorrect-command. Last accessed May 2020.

[3] Apple opens MultiPath TCP in iOS11. http://www.tessares.net/highlights-from-advances-in-networking-part-1.

[4] A service whereby MultiPath TCP attempts to use the lowest-latency interface. https://developer.apple.com/documentation/foundation/urlsessionconfiguration.multipathservicetype/2875971-interactive.

[5] Siri does more than ever. Even before you ask. https://www.apple.com/siri/, 2020.

[6] Augmented reality navigation overlays direction information onto the road. https://www.digitaltrends.com/cars/ar-navgation-hyundai-wayray-ces-2019/. Last accessed May 2021.

[7] Porsche, Hyundai invest in WayRay to make augmented-reality HUDs. https://www.cnet.com/roadshow/news/wayray-augmented-reality-hud-investment. Last accessed May 2021.

[8] Redefine your edge with a future-ready, intelligent mobile network architecture. httpJunes://www.business.att.com/content/dam/attbusiness/briefs/att-multi-access-edge-computing-brief.pdf, 2022. Last accessed June 2022.

[9] AWS left reeling after eight-hour DDoS. https://www.infosecurity-magazine.com/news/aws-customers-hit-by-eighthour-ddos/. Last accessed May 2020.

[10] Azure global outage: Our DNS update mangled domain records, says microsoft. https://www.zdnet.com/article/azure-global-outage-our-dns-update-mangled-domain-records-says-microsoft/. Last accessed May 2020.

[11] Azure Q&A. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable#q--a. Last accessed Mar 2022.

[12] Rajesh Krishna Balan, Darren Gergle, Mahadev Satyanarayanan, and James Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 272–285, 2007.

[13] Aruna Balasubramanian, Ratul Mahajan, and Arun Venkataramani. Augmenting mobile 3G using WiFi. In *Proceedings of the 8th International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 209–221, San Francisco, CA, June 2010.

[14] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *Proceedings of the 2011 ACM Conference on Computer Communications (SIGCOMM)*, 2011.

[15] Gaurav Banga, Peter Druschel, and Jeffry C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[16] Richard A Becker, Ramon Caceres, Karrie Hanson, Ji Meng Loh, Simon Urbanek, Alexander Varshavsky, and Chris Volinsky. Route classification using cellular handoff patterns. In *Proceedings of the 13th ACM international conference on Ubiquitous computing (Ubicomp)*, 2011.

[17] 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 Computer Communication Review (SIGCOMM)*, 44(3):87–95, 2014.

[18] Brendan Burns and David Oppenheimer. Design patterns for container-based distributed systems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2016.

[19] Panasonic Business. 5.5 4k camera road in thailand no 2. https://www.youtube.com/watch?v=F4bICvLY024. Last accessed May 2020.

[20] Vladimir Bychkovsky, Bret Hull, Allen Miu, Hari Balakrishnan, and Samuel Madden. A measurement study of vehicular internet access using in situ Wi-Fi networks. In *Proceedings of the 12th International Conference on Mobile Computing and Networking (MobiCom)*, 2006.

[21] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot–a technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[22] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[23] Yang Chen, Jens O Berg, Mostafa Ammar, and Ellen Zegura. Evaluation of data communication opportunities from oil field locations at remote areas. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (IMC)*, pages 117–126, 2011.

[24] Yung-Chih Chen, Yeon-sup Lim, Richard J Gibbens, Erich M Nahum, Ramin Khalili, and Don Towsley. A measurement-based study of MultiPath TCP performance over wireless networks. In *Proceedings of the 2013 Internet Measurement Conference (IMC)*, 2013.

[25] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *Proceedings of the IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 1789–1792, 2016.

[26] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems (EuroSys)*, pages 301–314, 2011.

[27] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[28] Under-provisioning: The curse of the cloud. `https://tinyurl.com/yrda7trn`, 2010. Last accessed Mar 2022.

[29] Connected Car Connectivity Growing As Cars Get Smarter. `https://www.spglobal.com/marketintelligence/en/news-insights/blog/connected-car-connectivity-growing-as-cars-get-smarter`.

[30] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 49–62, 2010.

[31] Eduardo Cuervo, Alec Wolman, Landon P Cox, Kiron Lebeck, Ali Razeen, Stefan Saroiu, and Madanlal Musuvathi. Kahawai: High-quality mobile gaming using gpu offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 121–135, 2015.

[32] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. WiFi, LTE, or both? Measuring multi-homed wireless internet performance. In *Proceedings of the 2014 Internet Measurement Conference (IMC)*, 2014.

[33] Pralhad Deshpande, Xiaoxiao Hou, and Samir R Das. Performance comparison of 3G and metro-scale WiFi for vehicular network access. In *Proceedings of the 2010 ACM Conference on Computer Communications (SIGCOMM)*, 2010.

[34] Pralhad Deshpande, Anand Kashyap, Chul Sung, and Samir R Das. Predictive methods for improved vehicular WiFi access. In *Proceedings of the 7th International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2009.

[35] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. *ACM SIGCOMM Computer Communication Review*, 41(4):362–373, 2011.

[36] BMW model upgrade measures taking effect from the summer of 2013. `https://www.press.bmwgroup.com/global/article/detail/T0141144EN/bmw-model-upgrade-measures-taking-effect-from-the-summer-of-2013`. Last accessed May 2021.

[37] Vincent Etter, Mohamed Kafsi, Ehsan Kazemi, Matthias Grossglauser, and Patrick Thiran. Where to go from here? Mobility prediction from instantaneous information. *Pervasive and Mobile Computing*, 9(6):784–797, 2013.

[38] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing web latency: the virtue of gentle aggression. In *Proceedings of the 2013 ACM Conference on Computer Communications (SIGCOMM)*, 2013.

[39] Complex event generation for business process monitoring using Apache Flink. `https://engineering.zalando.com/posts/2017/07/complex-event-generation-for-business-process-monitoring-using-apache-flink.html`. Last accessed March 2021.

[40] Real-time experiment analytics at Pinterest using Apache Flink. `https://medium.com/pinterest-engineering/real-time-experiment-analytics-at-pinterest-using-apache-flink-841c8df98dc2`. Last accessed March 2021.

[41] Behrouz A Forouzan. *TCP/IP protocol suite*. McGraw-Hill Higher Education, 2002.

[42] Armando Fox, Steven D Gribble, Eric A Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. *ACM SIGOPS Operating Systems Review (OSR)*, 30(5):160–170, 1996.

[43] Alexander Frommgen, Tobias Erbshäußer, Alejandro Buchmann, Torsten Zimmermann, and Klaus Wehrle. ReMP TCP: Low latency multipath TCP. In *IEEE International Conference on Communications (ICC)*, 2016.

[44] Matthew Furlong, Andrew Quinn, and Jason Flinn. The case for determinism on the edge. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, 2019.

[45] Gartner says four trends are shaping the future of public cloud. https://www.gartner.com/en/newsroom/press-releases 2021-08-02-gartner-says-four-trends-are-shaping-the-future-of-public-cloud. Last accessed September 2021.

[46] Gartner Says 5.8 Billion Enterprise and Automotive IoT Endpoints Will Be in Use in 2020. `https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io`, 2019.

[47] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The KITTI dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[48] Google Cloud in major global outage: Numerous services fail. `https://www.cbronline.com/news/google-cloud-down`. Last accessed May 2020.

[49] Google cloud status dashboard. `https://status.cloud.google.com/summary`. Last accessed May 2020.

[50] Jian Guo, Fangming Liu, Tao Wang, and John CS Lui. Pricing intra-datacenter networks with over-committed bandwidth guarantee. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[51] Yihua Ethan Guo, Ashkan Nikravesh, Z Morley Mao, Feng Qian, and Subhabrata Sen. Accelerating multipath transport through balanced subflow completion. In *Proceedings of the 23rd International Conference on Mobile Computing and Networking (MobiCom)*, 2017.

[52] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 357–371, 2018.

[53] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. You can teach elephants to dance: Agile VM handoff for edge computing. In *Proceedings of the 2nd IEEE/ACM Symposium on Edge Computing (SEC)*, 2017.

[54] Kiryong Ha, Padmanabhan Pillai, Wolfgang Richter, Yoshihisa Abe, and Mahadev Satyanarayanan. Just-in-time provisioning for cyber foraging. In *Proceedings of the 11th International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2013.

[55] Bo Han, Feng Qian, Shuai Hao, and Lusheng Ji. An anatomy of mobile web performance over Multipath TCP. In *Proceedings of the 11th ACM conference on Emerging networking experiments and technologies (CoNEXT)*. ACM, 2015.

[56] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. MP-DASH: Adaptive video streaming over preference-aware multipath. In *Proceedings of the 12th ACM conference on Emerging networking experiments and technologies (CoNEXT)*. ACM, 2016.

[57] Mark Handley, Olivier Bonaventure, Costin Raiciu, and Alan Ford. TCP extensions for multipath operation with multiple addresses. IETF RFC 6824, 2013.

[58] Joshua Hare, Lance Hartung, and Suman Banerjee. Beyond deployments and testbeds: experiences with public usage on vehicular WiFi hotspots. In *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2012.

[59] Brett D. Higgins, Kyungmin Lee, Jason Flinn, Thomas J. Giuli, Brian D. Noble, and Christopher Peplin. The future is cloudy: Reflecting prediction error in mobile applications. In *Proceedings of the 6th International Conference on Mobile Computing, Applications, and Services (MobiCASE)*, November 2014.

[60] Brett D. Higgins, Azarias Reda, Timur Alperovich, Jason Flinn, Thomas J. Giuli, Brian Noble, and David Watson. Intentional networking: Opportunistic exploitation of mobile network diversity. In *Proceedings of the 16th International Conference on Mobile Computing and Networking (MobiCom)*, pages 73–84, Chicago, IL, September 2010.

[61] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running Dive into the Future of Infrastructure*. O'Reilly Media, Inc., 1st edition, 2017.

[62] Michael R Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review*, 43:14–26, 2009.

[63] Michael R Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 51–60, 2009.

[64] Javascript HLS client using Media Source Extension. `https://github.com/video-dev/hls.js/`. Last accessed April 2021.

[65] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 95–110, 2018.

[66] Junxian Huang, Feng Qian, Yihua Guo, Yuanyuan Zhou, Qiang Xu, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. An in-depth study of LTE: Effect of network protocol and application behavior on performance. In *Proceedings of the 2013 ACM Conference on Computer Communications (SIGCOMM)*, 2013.

[67] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM Conference on Computer Communications (SIGCOMM)*, pages 187–198, 2014.

[68] Cisco VNI Forecast update. `https://www.ieee802.org/3/ad_hoc/bwa2/public/calls/19_0624/nowell_bwa_01_190624.pdf`, 2019. Last accessed May 2022.

[69] Big data on wheels . `https://www.ibmbigdatahub.com/blog/big-data-wheels`, 2014.

[70] Van Jacobson. Congestion avoidance and control. In *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, pages 314–329, Stanford, CA, August 1988.

[71] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with FESTIVE. In *Proceedings of the 8th ACM conference on Emerging networking experiments and technologies (CoNEXT)*, pages 97–108, 2012.

[72] Richard W.M Jones. Optimizing QEMU boot time. 2010.

[73] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: Measurements & analysis. In *Proceedings of the 2009 ACM Conference on Computer Communications (SIGCOMM)*, 2009.

[74] Ramin Khalili, Nicolas Gast, Miroslav Popovic, Utkarsh Upadhyay, and Jean-Yves Le Boudec. MPTCP is not pareto-optimal: Performance issues and a possible solution. In *Proceedings of the 8th ACM conference on Emerging networking experiments and technologies (CoNEXT)*. ACM, 2012.

[75] Byoungjip Kim, Jin-Young Ha, SangJeong Lee, Seungwoo Kang, Youngki Lee, Yunseok Rhee, Lama Nachman, and Junehwa Song. AdNext: a visit-pattern-aware mobile advertising system for urban commercial complexes. In *Proceedings of the 12th International Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 7–12, 2011.

[76] Matt Klein. Lyft's Envoy: Experiences operating a large service mesh. USENIX SREcon Americas, March 2017.

[77] KT's GiGA LTE. `https://www.ietf.org/proceedings/93/slides/slides-93-mptcp-3.pdf`.

[78] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 183–196, 2017.

[79] HyunJong Lee, Jason Flinn, and Basavaraj Tonshal. RAVEN: Improving interactive latency for the connected car. In *Proceedings of the 24th International Conference on Mobile Computing and Networking (MobiCom)*, 2018.

[80] Jinsung Lee, Sungyong Lee, Jongyun Lee, Sandesh Dhawaskar Sathyanarayana, Hyoyoung Lim, Jihoon Lee, Xiaoqing Zhu, Sangeeta Ramakrishnan, Dirk Grunwald, Kyunghan Lee, et al. PERCEIVE: deep learning-based cellular uplink prediction using real-time scheduling patterns. In *Proceedings of the 18th International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2020.

[81] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 151–165, 2015.

[82] Chi-Yu Li, Hsueh-Yang Liu, Po-Hao Huang, Hsu-Tung Chien, Guan-Hua Tu, Pei-Yuan Hong, and Ying-Dar Lin. Mobile edge computing platform deployment in 4G LTE networks: A middlebox approach. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*, 2018.

[83] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 104–120, 2017.

[84] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 467–483, 2016.

[85] Li Li, Ke Xu, Dan Wang, Chunyi Peng, Qingyang Xiao, and Rashid Mijumbi. A measurement study on TCP behaviors in HSPA+ networks on high-speed rails. In *Proceedings of the 34th Annual IEEE International Conference on Computer Communications (INFOCOM)*, 2015.

[86] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, page 122–1225, 2019.

[87] Yeon-sup Lim, Yung-Chih Chen, Erich M Nahum, Don Towsley, Richard J Gibbens, and Emmanuel Cecchet. Design, implementation, and evaluation of energy-aware multi-path TCP. In *Proceedings of the 11th ACM conference on Emerging networking experiments and technologies (CoNEXT)*. ACM, 2015.

[88] Yeon-sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. ECF: An MPTCP path scheduler to manage heterogeneous paths. In *Proceedings of the 13th ACM conference on Emerging networking experiments and technologies (CoNEXT)*. ACM, 2017.

[89] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common objects in context. In *European conference on computer vision (ECCV)*, pages 740–755. Springer, 2014.

[90] Tong Liu, Paramvir Bahl, and Imrich Chlamtac. Mobility modeling, location tracking, and trajectory prediction in wireless atm networks. *IEEE Journal on selected areas in communications*, 16(6):922–936, 1998.

[91] David E Lowell, Subhachandra Chandra, and Peter M Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, 2000.

[92] Lele Ma, Shanhe Yi, and Qun Li. Efficient service handoff across edge servers via docker container migration. In *Proceedings of the 2nd IEEE/ACM Symposium on Edge Computing (SEC)*, 2017.

[93] Ratul Mahajan, Jitendra Padhye, Sharad Agarwal, and Brian Zill. High performance vehicular connectivity with opportunistic erasure coding. 2012.

[94] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G Ives, and Sanjeev Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 693–708, 2017.

[95] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with Pensieve. In *Proceedings of the 2017 ACM Conference on Computer Communications (SIGCOMM)*, pages 197–210, 2017.

[96] The 5 biggest cloud computing trends in 2021. https://www.forbes.com/sites/bernardmarr/2020/11/02/the-5-biggest-cloud-computing-trends-in-2021/. Last accessed September 2021.

[97] Norman Maurer and Marvin Allen Wolfthal. *Netty in Action*. Manning Publications, 1st edition, 2015.

[98] Dirk Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.

[99] Microsoft's march 3 azure east us outage: What went wrong (or right)? `https://www.zdnet.com/article/microsofts-march-3-azure-east-us-outage-what-went-wrong-or-right/`. Last accessed May 2020.

[100] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 401–410, 2012.

[101] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 284–295. IEEE, 2005.

[102] Sam Newman. *Building Microservices*. O'Reilly, 2015.

[103] Anthony J Nicholson and Brian D Noble. Breadcrumbs: forecasting mobile connectivity. In *Proceedings of the 14th International Conference on Mobile Computing and Networking (MobiCom)*, 2008.

[104] Ashkan Nikravesh, David R Choffnes, Ethan Katz-Bassett, Z Morley Mao, and Matt Welsh. Mobile network performance from user devices: A longitudinal, multidimensional analysis. In *International Conference on Passive and Active Network Measurement (PAM)*, pages 12–22, 2014.

[105] Ashkan Nikravesh, Yihua Guo, Feng Qian, Z Morley Mao, and Subhabrata Sen. An in-depth understanding of Multipath TCP on mobile devices: Measurement and system design. In *Proceedings of the 22nd International Conference on Mobile Computing and Networking (MobiCom)*, 2016.

[106] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint-Malo, France, October 1997.

[107] Shadi Noghabi, Landon Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. The emerging landscape of edge-computing. *GetMobile: Mobile Computing and Communications*, 24, 2020.

[108] Shadi A Noghabi, John Kolb, Peter Bodik, and Eduardo Cuervo. Steel: Simplified development and deployment of edge-cloud applications. 2018.

[109] Jörg Ott and Dirk Kutscher. Drive-thru Internet: IEEE 802.11b for automobile users. In *Proceedings of the 23rd Annual IEEE International Conference on Computer Communications (INFOCOM)*, 2004.

[110] Jörg Ott and Dirk Kutscher. A disconnection-tolerant transport for Drive-thru Internet environments. In *Proceedings of the 24th Annual IEEE International Conference on Computer Communications (INFOCOM)*, 2005.

[111] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. Experimental evaluation of Multipath TCP schedulers. In *Proceedings of the 2014 ACM SIGCOMM workshop on capacity sharing*, 2014.

[112] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems*, 2009.

[113] Jeongyeup Paek, Kyu-Han Kim, Jatinder P Singh, and Ramesh Govindan. Energy-efficient positioning for smartphones using Cell-ID sequence matching. In *Proceedings of the 9th International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2011.

[114] Roger Pantos and William May. HTTP live streaming. IETF RFC 8216, 2017.

[115] Kushani Perera, Tanusri Bhattacharya, Lars Kulik, and James Bailey. Trajectory inference for mobile devices using connected cell towers. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 2015.

[116] Valerio Persico, Pietro Marchetta, Alessio Botta, and Antonio Pescapé. On network throughput variability in microsoft azure cloud. In *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*, 2015.

[117] sys — System-specific parameters and functions. `https://docs.python.org/3/library/sys.html#sys.settrace`. Last accessed Mar 2022.

[118] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 235–248, 2005.

[119] Yanyuan Qin, Shuai Hao, Krishna R Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. Abr streaming of vbr-encoded videos: characterization, challenges, and solutions. In *Proceedings of the 14th ACM conference on Emerging networking experiments and technologies (CoNEXT)*, pages 366–378, 2018.

[120] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govidan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services (MobiSys)*, June 2011.

[121] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with Multipath TCP. In *Proceedings of the 2011 ACM Conference on Computer Communications (SIGCOMM)*, 2011.

[122] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? Designing and implementing a deployable Multipath TCP. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[123] Matt Sargent, Vern Paxson, Mark Allman, and Jerry Chu. Computing TCP's retransmission timer. IETF RFC 6298, 2011.

[124] Mahadev Satyanarayanan, Nathan Beckmann, Grace A Lewis, and Brandon Lucia. The role of edge offload for hardware-accelerated mobile devices. *GetMobile: Mobile Computing and Communications*, 25(2):5–13, 2021.

[125] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 45–50, 2019.

[126] Salvatore Scellato, Mirco Musolesi, Cecilia Mascolo, Vito Latora, and Andrew T Campbell. Nextplace: a spatio-temporal prediction framework for pervasive systems. In *Proceedings of the 9th International Conference on Pervasive Computing*, 2011.

[127] Nicol N Schraudolph. A fast, compact approximation of the exponential function. *Neural Computation*, 11(4):853–862, 1999.

[128] Rahul Sharma and Avinash Singh. *Getting Started with Istio Service Mesh: Manage Microserves in Kubernetes*. Apress Springer, 2019.

[129] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bevier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proeceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[130] Joel Sommers and Paul Barford. Cell vs. WiFi: On the performance of metro area mobile connections. In *Proceedings of the 2012 Internet Measurement Conference (IMC)*, 2012.

[131] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. In *The 35th Annual IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–9, 2016.

[132] Sudarshan M Srinivasan, K Srikanth, Christopher R Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2004.

[133] Ya-Yunn Su and Jason Flinn. Slingshot: Deploying stateful services in wireless hotspots. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 79–92, 2005.

[134] Nik Sultana, Salvator Galea, David Greaves, Marcin Wójcik, Jonny Shipton, Richard Clegg, Luo Mai, Pietro Bressana, Robert Soulé, Richard Mortier, et al. Emu: Rapid prototyping of networking services. In *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference (ATC)*, pages 459–471, 2017.

[135] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 2003 Very Large Data Bases Conferences (VLDB)*, pages 309–320, 2003.

[136] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very Large Data Bases Conferences (VLDB)*, pages 159–170, 2007.

[137] David L Tennenhouse, Jonathan M Smith, W David Sincoskie, David J Wetherall, and Gary J Minden. A survey of active network research. *IEEE communications Magazine*, 35(1):80–86, 1997.

[138] Testing DASH and HLS streams on Linux. `http://ronallo.com/blog/testing-dash-and-hls-streams-on-linux/`. Last accessed May 2021.

[139] Eran Toch, Boaz Lerner, Eyal Ben-Zion, and Irad Ben-Gal. Analyzing large-scale human mobility data: a survey of machine learning methods and applications. *Knowledge and Information Systems*, 58(3):501–523, 2019.

[140] Introducing AthenaX, Uber engineering's open source streaming analytics platform. `https://eng.uber.com/athenax/`. Last accessed March 2021.

[141] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 374–389, 2017.

[142] Mobile Edge in Focus - Retail. https://www.verizon.com/business/verizonpartnersolutions/business/resources/solutionsbriefs/avidthink-aws-mobile-edge-in-focus-retail-research-brief-rev-c.pdf, 2021. Last accessed June 2022.

[143] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th ACM European Conference on Computer Systems*, 2015.

[144] Microsoft Azure proximity placement groups. https://docs.microsoft.com/en-us/azure/virtual-machines/co-location. Last accessed Mar 2022.

[145] Cedric Westphal, Stefan Lederer, Daniel Posch, Christian Timmerer, Aytac Azgin, Will Liu, Christopher Mueller, Andrea Detti, Daniel Corujo, Jianping Wang, et al. Information technology–dynamic adaptive streaming over HTTP (DASH)–part 1: Media presentation description and segment formats. ISO/IEC 23009-1: 2012, 2012.

[146] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for Multipath TCP. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[147] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, Mazin S Yousif, et al. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[148] wrk2. https://github.com/giltene/wrk2. Last accessed May 2020.

[149] Ying Xing, Stan Zdonik, and J-H Hwang. Dynamic load distribution in the borealis stream processor. In *21st International Conference on Data Engineering (ICDE'05)*, pages 791–802. IEEE, 2005.

[150] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[151] Fan Yang, Paul Amer, and Nasif Ekiz. A scheduler for Multipath TCP. In *22nd International Conference on Computer Communications and Networks (ICCCN)*, 2013.

[152] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2015.

[153] Wei Yu, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. A survey on the edge computing for the Internet of Things. *IEEE Access*, 6:6900–6919, 2017.

[154] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. Analyzing efficient stream processing on modern hardware. *Proceedings of the VLDB Endowment*, 12(5):516–530, 2019.

[155] Jianer Zhou, Qinghua Wu, Zhenyu Li, Steve Uhlig, Peter Steenkiste, Jian Chen, and Gaogang Xie. Demystifying and mitigating tcp stalls at the server side. In *Proceedings of the 11th ACM conference on Emerging networking experiments and technologies (CoNEXT)*, 2015.

[156] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the 9th ACM European Conference on Computer Systems*, 2014.