Flexible and Efficient Application-Networking Co-Design in Cloud Datacenters

by

Jingrong Chen

Department of Computer Science
Duke University

Defense Date: March 28, 2025

Approved:

Danyang Zhuo, Supervisor

Jeffrey Chase

Matthew Lentz

Lisa Wu Wills

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2025

ABSTRACT

Flexible and Efficient Application-Networking Co-Design in Cloud Datacenters

by

Jingrong Chen

Department of Computer Science
Duke University

Defense Date: March 28, 2025

Approved:

Danyang Zhuo, Supervisor

Jeffrey Chase

Matthew Lentz

Lisa Wu Wills

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2025

Abstract

Modern cloud datacenters support a wide range of distributed and data-intensive applications, from machine learning training to large-scale analytics and microservices. However, the networking abstractions available to these applications remain largely unchanged, continuing to expose a low-level, byte-stream interface that lacks awareness of application semantics. This rigid separation between applications and the network prevents cross-layer optimizations, forcing applications to either remain oblivious to network conditions or implement inefficient, ad hoc workarounds.

To address this gap, this dissertation explores pushing networking abstractions up the stack, enabling closer integration between application communication semantics and network infrastructure. Specifically, it investigates how cloud applications can leverage richer, intent-aware networking services rather than treating the network as an opaque transport layer. By doing so, we can bridge the divide between application logic and network behavior, improving efficiency, adaptability, and manageability in cloud environments.

We present two fully implemented systems and one forward-looking design proposal that exemplify this approach. NetHint enables applications to dynamically optimize data transfers by exposing real-time network hints, breaking the cloud's black-box networking model. It improves collective communication throughput by up to 2.7× by allowing applications to adapt to network conditions. mRPC rethinks Remote Procedure Call (RPC) handling by moving it into a managed OS service, eliminating redundant processing in per-application proxies and reducing RPC latency by up to 2.5×. Finally, NUSE explores a more flexible, hybrid kernel-userspace networking architecture, demonstrating how the OS itself can evolve to support diverse application-specific network processing needs. While NetHint and mRPC focus on immediate performance gains, NUSE represents a broader vision for making networking extensible in future cloud platforms.

Together, these contributions validate the core dissertation statement: that a closer integration between application communication semantics and network infrastructure is essential to improving efficiency and flexibility in cloud datacenters. By breaking traditional

abstraction barriers and enabling controlled information sharing across layers, we achieve significant gains in performance, manageability, and scalability, without sacrificing security or generality. By exposing richer network semantics and shifting key communication services into the system layer, we pave the way for more efficient, adaptable, and programmable cloud networking architectures.

Contents

Abstract . . . iv

List of Tables . . . ix

List of Figures . . . x

Acknowledgements . . . xii

1 Introduction . . . 1

1.1 Challenges Arising in Cloud and Large-Scale Applications . . . 1

1.2 Underlying Cause: The Byte-Level Interface Barrier . . . 2

1.3 Moving up the stack: High-Level Networking APIs Baked Into Systems . . . 3

1.4 Dissertation Statement . . . 4

1.5 Contributions . . . 4

1.6 Impact and Organization of the Dissertation . . . 5

2 NetHint: White-Box Networking for Multi-Tenant Data Centers . . . 7

2.1 Introduction . . . 7

2.2 Background . . . 11

2.2.1 Black-Box Networking Abstraction . . . 11

2.2.2 Adaptiveness in Data-Intensive Applications . . . 12

2.2.3 Addressing the Mismatch . . . 14

2.3 NetHint Overview . . . 16

2.4 Providing NetHint Service . . . 19

2.4.1 What Is in the Hint? . . . 19

2.4.2 Timely NetHint with Low Cost . . . 22

2.5 Adapting Transfer Schedules with NetHint . . . 23

2.5.1 Optimizing Collective Communication . . . 24

2.5.2 Optimizing Task Placement . . . 25

2.6 Flexible Adaptation for Stale Information . . . 27

2.7 Implementation . . . 28

2.8 Evaluation . . . 29
2.8.1 Setup and Workloads . . . 29
2.8.2 NetHint in Testbed Experiments . . . 31
2.8.3 NetHint in Simulations . . . 33
2.9 Discussion . . . 40
2.10 Related Work . . . 41
2.11 Summary . . . 42
3 Remote Procedure Call as a Managed System Service . . . 44
3.1 Introduction . . . 44
3.2 Background . . . 47
3.2.1 Remote Procedure Call . . . 48
3.2.2 The Need for Manageability . . . 50
3.3 Overview . . . 51
3.4 Design . . . 53
3.4.1 Dynamic RPC Binding . . . 53
3.4.2 Efficient RPC Policy Enforcement and Observability . . . 55
3.4.3 Live Upgrades . . . 59
3.4.4 Security Considerations . . . 60
3.5 Implementation . . . 62
3.6 Evaluation . . . 63
3.6.1 Microbenchmarks . . . 64
3.6.2 Efficient Policy Enforcement . . . 68
3.6.3 Live Upgrade . . . 69
3.6.4 Real Applications . . . 71
3.7 Related Work . . . 73
3.8 Summary . . . 75
4 NUSE: Towards a FUSE Counterpart for Networking . . . 76

4.1 Introduction 76
4.2 Background and Motivation 80
4.2.1 Limitations of Existing Approaches 80
4.2.2 Performance Implications of Shared Memory Queues 81
4.2.3 Motivation for a Hybrid-Kernel Approach 83
4.3 NUSE Design 83
4.3.1 NUSE Shim: Dispatching System Calls 84
4.3.2 NUSE System Service: Userspace Network Processing 84
4.3.3 NUSE Driver: Kernel-Space Virtual Device 85
4.3.4 UC Channel: Userspace NIC Offloading 85
4.4 Key Implementation Challenges 86
4.4.1 Balancing Performance and Application Transparency 86
4.4.2 Efficient and Robust Shared Memory Queue Design 87
4.4.3 Kernel Stack Integration and SKB Conversion 87
4.5 Applicability and Limitations 88
4.6 Related Work 89
4.7 Summary and Future Work 90
5 Conclusion 92
Bibliography 95
Biography 120

List of Tables

2.1 Notations and descriptions for NetHint. 19

2.2 Important factors related to the impact of staleness. 26

2.3 Testbed results: The system overhead of a NetHint server in CPU utilization, memory, and information collection latency. 32

3.1 mRPC Engine Interface. 62

3.2 Microbenchmark of small RPC latency: Round-trip RPC latencies for 64-byte requests and 8-byte responses. 66

3.3 Masstree analytics: Latency and the achieved throughput for GET operations. . 73

List of Figures

2.1 Applications have the ability and the incentive to adapt their transfer schedules based on network characteristics. 8

2.2 Examples to illustrate the black-box networking abstraction: tenants cannot predict their network performance. 11

2.3 Empirical allreduce (256MB) latency of 5 trials. 12

2.4 MapReduce jobs can adapt transfer schedules via task placement. 14

2.5 Allreduce can be performed with different topologies 15

2.6 NetHint overview. 17

2.7 Testbed results: Latency to compute transfer schedules. 31

2.8 Testbed results: NetHint's speedup for allreduce, broadcast, and mapreduce compared with user probing and not using network information. 32

2.9 Simulation results: Comparing NetHint with dynamic user probe in the default background traffic setting. 34

2.10 Simulation results for MapReduce: Extra overhead for MapReduce jobs comparing NetHint and user probing. 34

2.11 Simulation results for model serving: Using topology information alone can outperform using bandwidth information. 35

2.12 Simulation results: Average speedup to background traffic change period under two different topology settings. 37

2.13 Simulation results: NetHint's speedup to not using network information when we add noise to the input of NetHint. 38

2.14 Simulation results: NetHint's performance when varying the number of overlapped jobs. 38

2.15 Simulation results for distributed deep learning: NetHint's speedup to not using network information under different deployment environments. 39

2.16 Simulation results for distributed deep learning: Speedup for other fairness models. 40

2.17 Simulation results: NetHint's performance gain over perfect user probing. 40

3.1 Architectural comparison between current (RPC library + sidecar) and our proposed (RPC as a managed service) approaches. 45

3.2 Overview of the mRPC workflow from the perspective of the users (and their applications) as well as infrastructure operators. 49

3.3 Overview of memory management in mRPC. Shows an example for the `Get` RPC that includes a content-aware ACL policy. 57

3.4 Microbenchmark for large RPC goodput. 67

3.5 Microbenchmark for RPC rate and CPU scalability. 68

3.6 Efficient Support for Network Policies where RPC rates with and without policy are compared. 69

3.7 mRPC's ability to live upgrade under two scenarios. 70

3.8 Mean latency of in-application processing and network processing of microservices in DeathStarBench. 71

4.1 The NUSE architecture. 78

4.2 Message rate comparison between Native RDMA WRITE and NUSE (polling). . 81

4.3 Latency comparison between Native RDMA WRITE and NUSE (polling). 82

Acknowledgements

Five years of PhD life have flown by in an instant. Throughout this journey, I have almost always maintained a positive mindset, receiving invaluable encouragement and feedback along the way. I feel incredibly fortunate. None of this would have been possible without the unwavering support and help of the amazing people I have met.

First and foremost, I would like to express my deepest gratitude to my advisor, **Danyang Zhuo**. He is the best advisor I could have asked for—far exceeding my expectations from every aspect. A truly remarkable person is not just someone with outstanding abilities but someone whose presence and vision inspire and uplift those around them. He is such a person.

Danyang possesses exceptional wisdom and the ability to turn challenges into opportunities. His strong values, coupled with his relentless pursuit of the latest advancements in research, make him an extraordinary mentor. A great advisor provides guidance at all levels—offering direction on research problems, brainstorming ideas, and refining methodologies. But beyond that, Danyang also cares deeply about his students' growth, actively connecting us with external collaborators and internship opportunities. During the challenging early months of the pandemic in 2020, he provided much-needed support in both research and daily life. Words cannot fully express my gratitude. Thank you for shaping me into the researcher I am today. I hope that in the future, when you mention me to others, you will be proud of me.

I would also like to sincerely thank all the professors on my dissertation committee—Jeffrey Chase, Matthew Lentz, Lisa Wu Will, and Alvin Lebeck—for their guidance and valuable feedback on my dissertation.

Beyond my committee, I am deeply grateful to three professors who collaborated with me: Professor Xiaowei Yang, for her invaluable insights that greatly improved our work. While some of the suggested changes were challenging, I am ultimately grateful, as they made our paper much stronger; Tom Anderson and Ratul Mahajan, who were also Danyang's former advisors, provided numerous insightful comments on my work, even including some

unpublished ideas. Their feedback saved me significant effort and helped shape my research direction.

I am grateful to my labmates Xinhao Kong, Yongji Wu, Yicheng Jin, Yechen Xu, Jianxing Qin, and Alexander Du. In particular, I want to extend special thanks to Xinhao and Yongji.

Xinhao, to me, is a true confidant and source of inspiration. Having him around gave me the confidence to push through the challenges of PhD life. He set a high bar for alignment and excellence, making him both a role model and a close friend. We attended multiple NSDI conferences together, interned in Seattle at the same time, and shared countless experiences. Outside of research, he introduced me to the beauty of the U.S., and in times of personal struggles, our conversations were always enlightening and meaningful. I am truly grateful.

Yongji is my closest coauthor and a dear friend. He is one of the most hardworking people I have ever met (he also plays harder). I will always cherish the days we spent grinding through mRPC, reviewing each other's code, debugging late into the night, and preparing for meetings. It is hard to imagine ever having another peer with whom I will work so intensely and closely.

Yechen Xu and Jianxing Qin were excellent companions during the final stretch of my PhD. Our discussions on technical topics—especially around GPUs and system design—were always insightful and fun. I truly enjoyed and appreciated having them around.

Beyond my lab, I would also like to thank many fellow students at Duke, including Jiyao Hu, Xiao Zhang, Shihan Lin, Shujun Qi, Jiaao (Mason) Ma, Ceyu (Entropy) Xu, Yuxi Liu, Yiheng Shen, Zonghao Huang, Fangzhu Shen, Haibo Xiu, Yihao Hu, Yanping Zhang, Yujie Zhang, Luka Duranovic, Yongkang Li, Yi Zhou, Yalu Cai, Zhengjie Miao, Kangning Wang, Hanrui Zhang, Chenwei Wu, and Guozhen She. Jiyao and Xiao, thank you for all the help and support during my early PhD days. Ceyu and Jiaao, thank you for all the great times we shared. Yuxi and Yiheng, thank you for being my table tennis partners—without you, my PhD would have been entirely devoid of sports! Shihan Lin, a special mention. We go way back to our undergraduate days in Professor Yang Chen's lab, along with Jiyao. Shihan started his PhD one year before me and is graduating alongside me, making him a constant

presence throughout my journey. In addition to being a crucial coauthor on mRPC, he has been a true friend, offering both technical and personal support. I deeply appreciate everything.

Even though not at Duke, I owe a special thank you to Xiangfeng Zhu, who not only interned with me but also co-authored a paper together. While I was in Seattle, he introduced me to bouldering, board games, and provided immense help in my daily life. I am extremely grateful for his kindness.

I would also like to acknowledge my professors at Duke—Danfeng Zhang, Tingjun Chen, and Bruce Maggs. As my PhD coursework gradually came to an end, I truly appreciated the opportunity to learn from them in my final years as a student.

A big thank you to Liz Labriola and the graduate administration team at Duke for their invaluable support in handling various administrative matters.

During my PhD, I had the privilege of interning at different places, each providing invaluable experiences. At Uber, I was fortunate to work under Hongqiang (Harry) Liu, an exceptionally resourceful and insightful mentor whom I deeply respect. His guidance and support have been instrumental, and I feel honored that Uber will be my first workplace after graduation. At Meta, I worked with Liang Luo, who was also a coauthor on my first publication. He introduced me to the field of MLSys research, one of the most exciting areas today. His humor and mentorship made a lasting impact on my research journey.

I am also immensely grateful to Hong Zhang and Wei Bai, both coauthors and research seniors, with whom I share an academic lineage from HKUST SingLab. Our collaboration was particularly meaningful, and I deeply appreciate their insights and support. I would also like to acknowledge Shuihai Hu and Li Chen. Our reunion in Sydney in 2024 brought back many fond memories.

I am deeply grateful to Yingcheng Wang, whose love and support sustained me throughout this journey. Thank you for your patience and belief in me carried me through it all.

Finally, my deepest gratitude goes to my family—my parents and grandparents, for their unwavering love and support.

1. Introduction

The interface between application and networking has significantly evolved over the past decades. Early distributed applications relied on general-purpose, low-level socket interfaces for communication, which offered a simple but primitive abstraction of sending and receiving bytes over the network. This model gave developers flexibility but forced them to manage many details of communications (e.g., message framing, reliability, load balancing) on their own. As systems became more complex and performance-demanding, higher-level *workload-specific networking API* has emerged—for example, Remote Procedure Call (RPC) frameworks for microservices, object-oriented key-value stores for distributed state management, and collective communication for synchronizing gradients and parameters in recent learning systems. These specialized APIs encapsulate common communication patterns, easing developer burden and often improving efficiency for their target workloads. This evolution reflects a broader trend: moving *up the stack* to provide developers with abstractions that match their application semantics, rather than the low-level mechanics of networks.

To bridge the gap between high-level networking primitives and the low-level byte-oriented networking interfaces, the classic approach has been using layered abstraction: build higher-level communication libraries on top of low-level socket `send/recv` or IBVerbs primitives. For example, RPC libraries marshal object into bytes, and frameworks assemble complex communication patterns from simple message sends. This layer-by-layer design proved effective in early networked systems, hiding complexity behind clean APIs without modifying underlying network mechanisms. However, as we enter the cloud era, and as greater needs rise in flexible application traffic management and performance, this traditional approach is facing challenges.

1.1 Challenges Arising in Cloud and Large-Scale Applications

As applications migrate to cloud environments and scale to thousands of interconnected components, the old model of strictly layered networking abstractions is breaking down.

Two key challenges have emerged:

- **Opaque network abstraction in the cloud.** Cloud providers typically present the network to tenants as a black-box service, revealing little about underlying conditions. Applications running in public clouds cannot see vital network information, such as current congestion, topology, or the presence of competing traffic, through standard socket APIs. High-level libraries built on these opaque abstractions cannot easily adapt to network variability, often leading to suboptimal performance. In essence, distributed applications must make data transfer decisions without insight into the network's state, which can result in inefficient use of bandwidth and increased tail latencies [46].
- **RPC traffic management at scale.** When an application is composed of many microservices, the need of gaining greater visibility and control over RPC communication grows. The conventional solution ties and RPC stub library into each application and then deploys sidecar proxies (or a service mesh) alongside each service to enforce policies (load balancing, security, monitoring). This adds substantial overhead and complexity [45]. The sidecar proxy ends up re-inspecting and modifying RPC messages that the application already assembled, incurring redundant (un)marshalling. Moreover, implementing global policies (like rate limiting across multiple applications) in a decentralized way across proxies is cumbersome, and upgrades to new network features (e.g., RDMA or high-performance NIC capabilities) require upgrading every microservice's RPC library. In short, the current RPC management model struggles to scale efficiently—it sacrifices performance and makes consistent policy enforcement difficult.

1.2 Underlying Cause: The Byte-Level Interface Barrier

The fundamental root cause of these conflicts is the narrow, *byte-level* interface between applications and the network system. Today's operating systems and transport protocols are built to send and receive raw bytes; any higher-level meaning of those bytes is known only to the application. This strict separation makes cross-layer cooperation or optimization

nearly impossible. The system cannot optimize data transfers or scheduling beyond generic strategies (e.g., FIFO packet queues, general congestion control algorithms) because it lacks *application semantics*; it does not know which streams correspond to high-priority RPCs or which flows are performing an all-to-all shuffle for a big data job. Conversely, the application cannot make fully informed decisions because it is blind to the *system and network state*; it does not know if the network is congested, which path its traffic takes, or how resrouces are shared with co-located services.

In traditional settings, this separation was an acceptable trade-off, as applications ran on relatively static infrastructure or controlled, single-tenant clusters where either conservative, one-size-fits-all network policies sufficed, or customization can be made with relatively low effort. Further, applications were often less sensitive to communication efficiency, as performance bottlenecks typically lay elsewhere, such as in computation or storage.

In cloud datacenters, however, the cost of the "dumb pipe" approach is much higher. Today's could applications become increasingly distributed by design, often structured as microservices or large-scale prarllel jobs that rely heavily on fine-grained RPCs or collective communications. Without some form of information exchanges or coordination across the application-network boundary, we miss opportunities for optimization (as seen in the black-box cloud networking issue) and we incur unnecessary overheads (as seen in the RPC/sidecar issue). These inefficiencies point to a need for rethinking the division of responsibilities between applications and the networking stack.

1.3 Moving up the stack: High-Level Networking APIs Baked Into Systems

Given these challenges, a natural question arises: what if the system provided high-level, intent-based networking APIs to applications, rather than just byte streams? In other words, suppose cloud applications could directly convey their communication intents (such as "broadcast this dataset" or "call this remote function with policy X") to the underlying system, and in return obtain insight or assistance from the network. How would such applications be designed, and how would such a system be structured?

This dissertation explores these questions. Elevating the application-network interface requires addressing two sides of the co-design coin: (1) Application adaption: applications would need to make use of richer network semantics, scheduling transfers or structuring communication with awareness of network conditions and services. (2) System design: the operating system and networking stack would need a flexible architecture capable of support diverse high-level APIs (for RPCs, collective communications, etc.) efficiently and safely. The system must expose more information or services to applications without compromising security isolation or performance. Achieving this is a balancing act: we seek to retain the *flexibility* of user-level libraries (which can evolve rapidly and crate to application needs) while obtaining the *efficiency* and centralized control of kernel-level mechanisms.

1.4 Dissertation Statement

In this dissertation, I argue that improving communication efficiency without compromising generality and security in cloud datacenters calls for a redesign of the interface between applications and networking systems, both in what information and abstraction is exposed across it, and in where key communication logic should reside.

1.5 Contributions

To support this dissertation statement, the dissertation introduces and evaluates three synergistic systems that bridge the gap between application-level intent and network-level mechanism:

NetHint A system that breaks the network black-box by providing applications with network *hints*. NetHint establishes an interactive feedback loop between cloud tenants and the provider: the cloud shares indirect but useful network information (such as topology of the tenant's virtual machines, current bandwidth usage, and locations of congestions), and applications use these hints to schedule data transfers more intelligently. By exposing selected network internals to data-intensive applications, NetHint allows cross-layer optimizations that was previously impossible. For example, an deep learning traning job can choose and all-reduce ring/tree that avoids congested links. The NetHint design supplies rich

network context with minimal overhead on the provider side, and it improves application-level throughput and completion times (e.g., speeding up collective operations by up to 2.7× in experiments).

mRPC A re-imagination of RPC as a *managed OS service* rather than an application-level library. mRPC elevates RPC handling into operating system, where a dedicated system service takes over tasks like argument marshalling, message routing, and policy enforcement. By doing so, we eliminate the redundant work done by sidecar proxies and enable a single, authoritative point for applying policies. Applications still define their RPC interfaces as usual, but the heavy-lifting of serialization and applying network policies (load balancing, encryption, rate limits, etc.) happens within mRPC's domain. This co-design drastically reduce per-call overhead. For instance, mRPC speeds up a standard microservice benchmark by up to 2.5× compared to the conventional approach that uses user-library with a sidecar. Meanwhile, it simplifies management, as updates to RPC handling (e.g., adopting RDMA for transport or changing a security policy) can be deployed in one place and immediately benefit all applications.

NUSE NUSE demonstrates that an operating system can be made extensible even in its data-plane APIs without incurring large performance penalties, thereby providing a path to support diverse application-specific network primitives and host-networking innovations inside the system. This contribution generalizes the lessons of the earlier case studies, examining how far "up the stack" we can push networking services in a cloud operating system, and what the costs and benefits are under real workloads.

1.6 Impact and Organization of the Dissertation

Pushing networking abstractions closer to application needs—moving up the stack—is a powerful strategy to resolve the tension between applications and cloud networking infrastructure. The contributions of this dissertation validate this idea across different levels of system design. NetHint and mRPC serve as concrete case studies, demonstrating that targeted application-network co-design can yield immediate gains in performance and manage-

ability. NetHint's approach to exposing network hints has already influenced other research efforts, such as its simulator being utilized in MCCS [281], showing broader applicability beyond the scope of this dissertation. mRPC, by shifting RPC management into the OS, simplifies policy enforcement and reduces overhead, providing an alternative to sidecar-based architectures.

NUSE broadens the perspective by exploring a flexible OS architecture for networking, proposing a hybrid user-kernel approach that could enable greater adaptability in cloud systems. While NUSE remains a design proposal rather than a fully implemented system, it lays the groundwork for future research in modular and extensible network processing frameworks. These results collectively show that while the desirability of richer, cross-layer interfaces is universal, the means of achieving them can vary—some cases benefit from exposing selective network information, while others may require fundamental shifts in system architecture.

The rest of this dissertation is organized as follows. Chapter 2 presents the design, implementation, and evaluation of NetHint, detailing how hint generation and consumption work, and quantifying its benefits for various data-intensive workloads. Chapter 3 details mRPC, describing the system architecture, and experimental results comparing it to conventional RPC frameworks. Chapter 4 introduces NUSE as a forward-looking design proposal, exploring the feasibility of a hybrid user-kernel networking architecture to enable greater flexibility in cloud networking. While not fully implemented, it outlines key challenges, potential benefits, and directions for future research in modular network processing. Finally, chapter 5 concludes the dissertation, summarizing the key findings, discussing their broader implications, and identifying open research questions and opportunities for future work in application-network co-design.

2. NetHint: White-Box Networking for Multi-Tenant Data Centers

To bridge the gap between application needs and network capabilities, we must rethink the network abstraction exposed to cloud tenants. Instead of treating the network as a black box, an alternative approach is to selectively expose useful network hints that enable applications to optimize their communication patterns. This leads to the design of NetHint, a system that provides applications with real-time insights into network topology and congestion, allowing them to dynamically adjust data transfers for improved efficiency. The next chapter introduces NetHint in detail, demonstrating how a white-box networking approach can significantly enhance the performance of data-intensive cloud applications.

2.1 Introduction

Data-intensive applications (e.g., network functions, data analytics, deep learning) have increasingly moved to the cloud for resource elasticity, performance, security, and ease of management. The performance of the cloud network is critical for these applications' performance. Cloud providers have thus spent significant effort to optimize various aspects of cloud networks, including network topology [238, 104, 245], congestion control and network stack [229, 7, 304, 135, 128, 102, 264], load balancing [6, 142, 292, 201], bandwidth guarantee [22, 129, 154, 158, 218, 10], debugging [14, 98], fault recovery [169], hardware [20, 85, 181, 168], and virtualization [213].

Today, a cloud provider exposes the network to its tenants as a black box: the cloud tenants have little visibility into their expected network performance (e.g., a constant worst-case bandwidth assurance) or the underlying network characteristics including the link-layer network topology, number of co-locating tenants, and instantaneous available bandwidth.

The black-box model has worked well for decades due to its simplicity. However, with the emergence of popular data-intensive applications (e.g., data analytics, distributed deep learning, and distributed reinforcement learning) in the cloud, we observe that such a black-box model is no longer efficient (§3.2). The crux is that many of these emerging applications

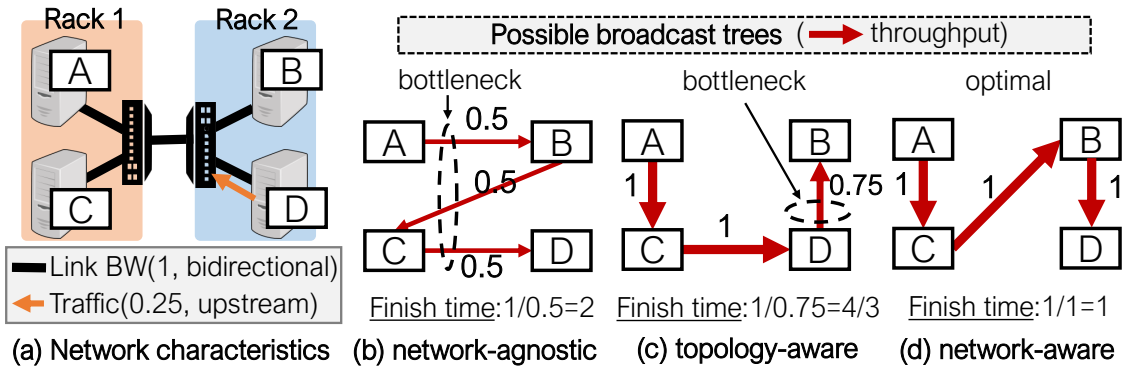


FIGURE 2.1: **Applications have the ability and the incentive to adapt their transfer schedules based on network characteristics:** Consider broadcasting a unit-size data object from VM A to VM B, C, and D. (a) shows the network characteristics, all links have bidirectional bandwidth of 1. VM D has upstream background traffic of 0.25. (b) to (d) show possible broadcast trees and their corresponding broadcast finish time. The arrows represent traffic flows and the numbers represent the throughput.

have both the *ability* and the *incentive* to adapt their transfer schedules based on the underlying network characteristics, but it is difficult to do so with a black-box network.

Consider broadcast, an important communication primitive in reinforcement learning and ensemble model serving. Figure 2.1 shows an example that VM A broadcasts to VM B to VM D. Figure 2.1b shows a possible broadcast tree constructed under the black-box model. Without the underlying network characteristics, the broadcast tree is *network-agnostic*, which introduces link stress on the cross-rack link. Figure 2.1c shows a broadcast tree based on the topology information (i.e., topology-aware), which improves the broadcast finish time from 2 to $\frac{4}{3}$ time units by minimizing the cross-rack traffic. Figure 2.1d shows a broadcast tree based on both the topology and bandwidth information (i.e., network-aware). It builds an optimal broadcast tree that avoids the congested upstream link on VM D, further improving the finish time to 1 time unit. The performance gains increase for data center networks that have larger oversubscription ratios or more skewed traffic.

The above example illustrates a fundamental *mismatch* between the black-box nature of existing network abstractions and the ability of a data-intensive application to adapt its traffic. With the black-box model, the cloud tenant is unaware of the network char-

acteristics, and the cloud provider is unaware of the application communication semantics and the transfer schedule. This misses an opportunity for the cloud tenants and the cloud provider to adapt the data flows to the underlying network topology and conditions to enhance performance and efficiency for these applications. The potential gains are substantial: our benchmark experiment on AWS shows that the allreduce latency for a deep learning experiment varies by up to 2.8× across different allreduce transfer schedules. One candidate approach is for applications to probe and profile the network and then plan their data flows accordingly [176, 9]. A second option is to report their possible transfer schedules to the provider for the provider to choose. We observe that these alternatives introduce substantial communication latency and system overhead (§2.2.2).

In this chapter, we explore a *white-box* approach to resolve this mismatch. One possibility would be for the cloud provider to expose the physical network topology, the VM locations, along with bandwidth assurances to the application. However, this approach has two major drawbacks. First, exposing VM placement and data center network topology may compromise security for cloud tenants and can raise concerns for the cloud provider (§3.2). Second, the bandwidth available to a tenant depends on the communication patterns of other tenants, which may be highly dynamic. Predictions that are not timely or not accurate may do more harm than good.

This study explores an alternative approach. We design and implement NetHint, a mechanism for a cloud tenant and cloud provider to interact to enhance the application performance jointly. The key idea is that the provider provides a *hint* — an *indirect indication* of the bandwidth allocation to a cloud tenant (e.g., a virtual link-layer network topology, number of co-locating tenants, network bandwidth utilization). The tenant applications then adapt their transfer schedules based on the hints, which may change over time. NetHint balances confidentiality and expressiveness: on one hand, the hint avoids exposing the physical network topology or traffic characteristics of other tenants (§4.5). On the other hand, we show that the hint provides sufficient network information to enable tenants to plan efficient transfer schedules. (§2.5).

The effectiveness of NetHint relies on addressing three important challenges. First, *what information should the hint contain?* We provide each cloud tenant with a virtual link-layer network topology along with available bandwidth on each link in the virtual topology. This allows applications to adapt their transfer schedules to avoid network congestion.

The second challenge is *how to provide this hint at a low cost.* We design a two-layer aggregation method to collect network statistics on the hosts. We designate a NetHint server in a rack to aggregate network characteristics in the rack. NetHint servers then use all-to-all communication to exchange network characteristics globally. A cloud tenant can thus query its rack-local NetHint server for hints.

The final challenge is *how should applications react to the hint.* We present several use cases for NetHint to optimize communication in a range of popular data-intensive applications including deep learning, MapReduce, and serving ensemble models. The takeaway is that for all these applications, tenants can use the NetHint information via simple scheduling algorithms. Adaptation also has a downside: hints can be stale and adapting transfer schedules based on stale information can hurt performance. We design a policy for applications to adapts flexibly with different hints in different scenarios: applications use temporal bandwidth information when background network conditions are stable and adaptation overhead is low, and otherwise applications fall back to using only the time-invariant topology information (§2.6).

We evaluate the overheads and the potential performance gain of having NetHint in data centers using a small testbed and large-scale simulations. Our results show that NetHint speeds up the average performance of allreduce completion time in distributed data-parallel deep learning, broadcast completion time in ensemble model serving, and MapReduce shuffle completion time in distributed data analytics by 2.7×, 1.5×, and 1.2×, respectively. Moreover, these benefits are cheap to obtain: NetHint incurs modest CPU, memory, and network bandwidth overheads.

In summary, this study makes the following contributions:

- We identify a mismatch between the current black-box network abstraction and the com-

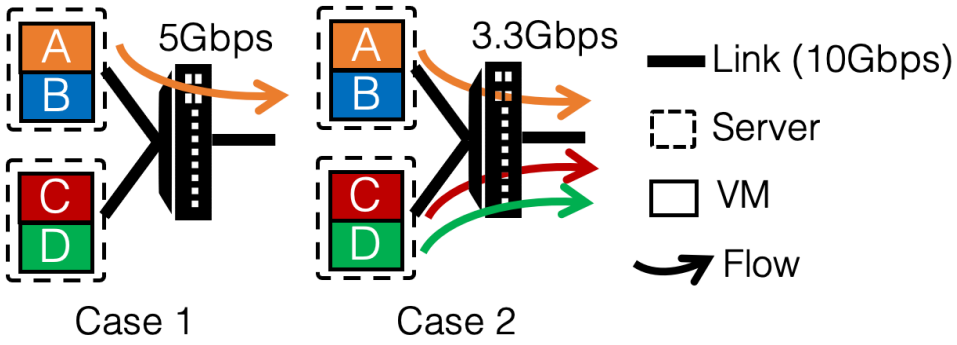


FIGURE 2.2: **Examples to illustrate the black-box networking abstraction: tenants cannot predict their network performance.** VM A to D are placed in two servers. All links have 10 Gbps bandwidth. We assume bandwidth is statically partitioned on the end host (each VM can get at most 5 Gbps).

munication needs of data-intensive applications.

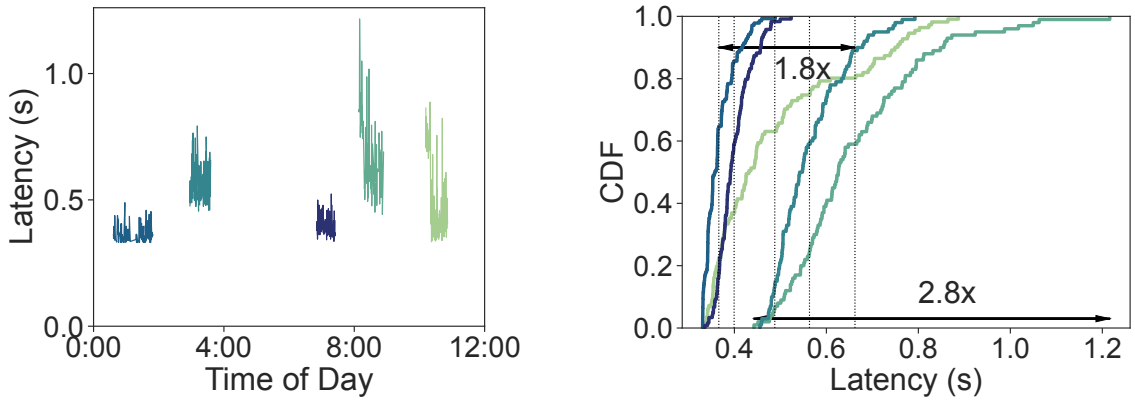
- We explore a white-box networking approach for multi-tenant data centers.
- We design and implement NetHint, a low-cost system to allow data-intensive applications to adapt their data transfer schedules to enhance performance.

2.2 Background

2.2.1 Black-Box Networking Abstraction

Today, the networking abstraction a cloud has is merely a per-VM bandwidth allocation at the end hosts. The abstraction is a *black box*: tenants are unaware of the underlying network characteristics including network topology, number of co-locating tenants, and instantaneous available bandwidth. As a result, the cloud tenants cannot predict their network performance. Figure 2.2 shows an example. Even with a static allocation of 5 Gbps per VM, VM A cannot predict its network performance because it depends on the traffic demand of other VMs. VM A can get only a bandwidth of 3.33 Gbps when two flows of VM C and D cause congestion inside the network (case 2). Even with work-conserving bandwidth guarantees, a VM's network performance depends on other VMs.

To quantify this effect, we benchmark allreduce latency on Amazon Web Service (AWS). Allreduce is a collective communication primitive that is commonly used for distributed deep



(a) Allreduce latency across time

(b) CDF

FIGURE 2.3: Empirical allreduce (256MB) latency of 5 trials. Two trials may have different VM allocations spatially, and each trial contains 100 consecutive runs. (a) shows 5 trials over different times of a day. In (b), each line is the latency CDF of a trial. Each vertical line is the mean latency for a trial. Allreduce latencies vary both across time (up to 2.8x) and across VM allocations (up to 1.8x).

learning. It aggregates a vector (i.e., gradient updates in deep learning) across all worker processes (each running in its own VM). In our experiment, we launch 32 g4dn.2XL (with Linux kernel 5.3) instances in the EC2 US-East-1 region and test ring-allreduce latency with NVIDIA NCCL (version 2.4.8)—the most popular collective communication library for deep learning—for 100 consecutive runs. We repeat the above experiment for 5 trials, and different trials may have different VM placements on the physical topology. Figure 2.3 shows our findings: ring-allreduce performance on 256MB buffer varies both spatially across different trials and temporally within a trial. Comparing across different trials, the fastest trial has a 1.8× better mean performance than the slowest trial; comparing the 100 runs within a trial, the fastest run is up to 2.8× faster than the slowest run.

2.2.2 Adaptiveness in Data-Intensive Applications

Besides reinforcement learning and ensemble model serving, which can broadcast model and input data adaptively, as illustrated in Figure 2.1, we show that many other applications also have both the ability and incentive to adapt their transfer schedules based on the underlying network characteristics.

Many distributed data analytics workloads contain network-intensive shuffle phases between different job stages. For example, the shuffle in MapReduce applications creates an all to all data transfer between the map and reduce stages. The shuffle phase accounts for a large portion of the execution time for many data analytics workloads [53], and numerous studies [53, 52, 291, 287, 298, 119, 8] have demonstrated that optimizing shuffle performance significantly improves application performance. Given network characteristics, distributed data analytics applications can change their transfer schedules (by changing the task placement) to minimize shuffle completion time. Figure 2.4a shows the shuffle traffic for a MapReduce job with two mappers (m1 and m2) and two reducers (r1 and r2). We observe from Figure 2.4b to Figure 2.4d that allocating mappers and reducers based on the topology and bandwidth information effectively improves this shuffle completion time from 4 to 2 units. Moreover, emerging task-based distributed systems (e.g., Ray, Dask, Hydro) support applications with dynamic task graphs. Similar to the MapReduce example, we can change the transfer schedule of these applications by choosing different VMs to place a task.

Moreover, many deep learning jobs are network-intensive. This claim is validated by numerous recent studies [49, 290, 236, 125, 105] and observations from production clusters (e.g., Microsoft [282, 96, 126] and ByteDance [210]). In particular, as mentioned in §2.2.1, deep learning jobs contain an allreduce phase to synchronize gradient updates among workers in each training iteration. As shown in Figure 2.5, an allreduce phase has multiple candidate topologies. For example, the allreduce traffic can be sent via a ring connecting all the workers with a flexible ordering (Figure 2.5a and Figure 2.5b). Or, we can build an allreduce tree to (1) aggregate gradient updates to one of the workers, and (2) send the aggregated gradient updates back in the reverse direction (Figure 2.5c and Figure 2.5d). Different allreduce topologies introduce different transfer schedules. Thus, given network characteristics, deep learning jobs can change their transfer schedules by selecting the algorithm and configuration of allreduce.

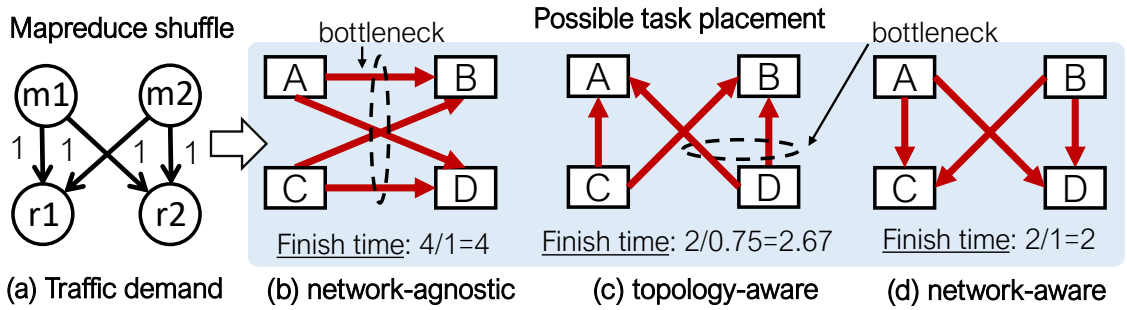


FIGURE 2.4: **MapReduce jobs can adapt transfer schedules via task placement.** Assume the same network characteristics as in Figure 2.1a. (a) shows the traffic demand for a MapReduce shuffle. Each arrow represents a unit traffic. (b) to (d) show possible task placement and the corresponding shuffle finish time.

2.2.3 Addressing the Mismatch

The black-box nature of the existing networking abstraction and the adaptiveness of data-intensive applications create a mismatch. Data-intensive applications would benefit from more network information from the cloud provider to configure their transfer schedules, but black-box networking hides this information.

Solutions based on the black-box abstraction. There are two approaches to address this mismatch without modifying the existing black-box networking abstraction. One possible approach is to let the cloud provider optimize the communication for tenants as a cloud service. To this end, we first have to develop a general networking API for cloud tenants to express their communication semantics, traffic loads and optimization objectives to the cloud provider. The API design should be similar to the coflow abstraction [53] or the virtual cluster abstraction [22], but more general to support a large variety of possible traffic patterns and user-defined objectives. Moreover, a recent measurement study [265] shows that major public clouds exhibit high bandwidth variability at a time granularity of seconds. Thus it is hard, if not impossible, for the cloud provider to perform timely network scheduling for thousands of tenants in a centralized manner, while ensuring network SLAs (e.g., defined via the networking API) for each tenant respectively.

Another potential approach is for cloud tenants to run extensive performance profiling

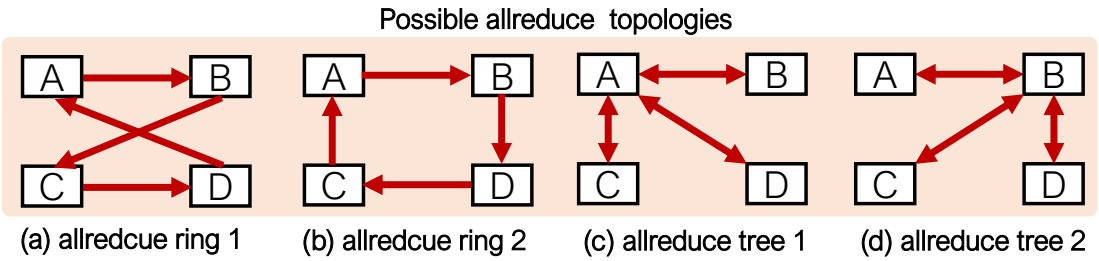


FIGURE 2.5: **Allreduce can be performed with different topologies.** (a) to (d) show 4 possible allreduce topologies to perform allreduce among the 4 VMs (workers).

in their allocated VMs [176, 89, 155]. For example, PLink [176] probes the VM pair-wise bandwidth and latency with DPDK and uses K-means clustering to reverse engineer the underlying network topology. This allows it to achieve high allreduce performance by choosing a good allreduce algorithm. Choreo [155] uses 3-step measurements to pinpoint congested links in the data center network to schedule data analytics workloads. Similar approaches were explored decades ago on Internet traffic routing on wide-area overlay networks [9]: picking a high-performance Internet path based on user measurement. Unfortunately, this approach is both costly, as each tenant/user has to profile the network independently, and slow, because the probing phase delays the start of the application. The PLink authors told us that they use 10000 packets to determinte bandwidth between a pair of hosts. Choreo generates 3 minutes of probe traffic to infer the network characteristics for 10 VMs.

A white-box network abstraction? Given the deficiencies of the two black-box based approaches, we instead explore a white-box approach: the provider reveals essential information about the network characteristics to the tenant, and the tenants then optimize their transfer schedules accordingly.

One possible way to achieve this objective is for the cloud provider to reveal to a tenant the location of each VM in the physical link-layer network topology, and estimate available bandwidth between each of the VM-pairs. However, this method can raise security and competitive issues. First, exposing VM allocations in the physical network introduces privacy risks for cloud tenants. For example, a malicious user can locate a targeted tenant's VMs

and perform attacks. Second, the exposed VM allocation information can raise competitive concerns for the cloud provider. For instance, this information might be valuable for competitors to learn a cloud provider's scheduling policies, thus, lowering its competitive advantage. Third, the bandwidth a tenant can acquire depends on the transfer schedules of *all* the tenants, and a single change in transfer schedule of one tenant may trigger a recalculation for all the tenants. As such, it is computationally expensive for the cloud provider to update the bandwidth shares in real time. Moreover, an application's bandwidth also depends on *its own* transfer schedule. For example, in Case 2 of Figure 2.2, if VM A sends one extra flow, the total egress bandwidth of VM A increases to 5 Gbps[1]. As a result, without knowing a tenant's transfer schedule, the cloud provider cannot provide accurate bandwidth estimates to its tenants.

2.3 NetHint Overview

NetHint is an interactive mechanism between a cloud tenant and a cloud provider to jointly enhance the application performance. The key idea is that the provider provides a *hint* — an *indirect indication* of the underlying network characteristics (e.g., a virtual link-layer topology for a tenant's VMs, number of co-located tenants, network bandwidth utilization) to a cloud tenant. As illustrated in Figure 2.6, the provider provides a NetHint service, which periodically (100 ms by default) collects the hint information to capture changes of the underlying network characteristics. A tenant application can query the NetHint service to get the hint information, and then adapt its transfer schedules based on this provided hint. Note that NetHint does not change the fairness mechanism of the underlying network. A tenant can opt in/out any time — whether or not to use NetHint will *not* affect its fair share of the network.

The hint provides a white-box network abstraction which includes additional network information to tenants. As such, users can infer their best transfer schedule without substantial probing latency or communication overhead with the provider. The hint exposes neither the

[1] Assume per-flow fair sharing in the network.

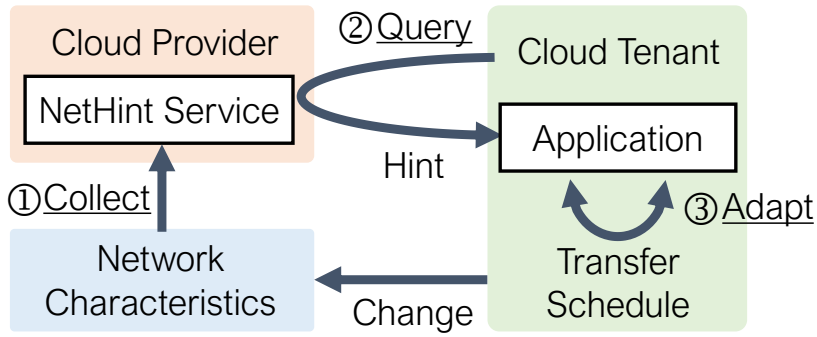


FIGURE 2.6: NetHint overview. NetHint service collects network characteristics. Cloud tenants poll hints from NetHint service and adapt their transfer schedules.

physical network topology nor the location of a tenant's VMs within it (e.g., which racks). Compared with providing bandwidth information, the hint relieves the provider from the burden of calculating accurate bandwidth allocations. Moreover, compared with calculating bandwidth allocation, it is easier to acquire accurate hint messages (e.g., a virtual link-layer topology for a tenant's VMs, number of co-located tenants, network bandwidth utilization). As such, the provider is free from the potential risk of providing inaccurate information.

We require NetHint to be: (1) *readily deployable:* all the mechanisms are implementable using commodity hardware; (2) *low cost:* the cloud provider can collect network characteristics with minimal CPU, memory, and bandwidth overheads; (3) *useful:* data-intensive workloads can leverage the hints to achieve high performance. To achieve these goals, NetHint's design and implementation must address three questions. First, what hints should be provided to the tenants? Second, how should cloud providers collect the hints with low cost? Third, how should applications use the hints to adapt their transfer schedules?

NetHint describes a virtual link-layer topology that connects a tenant's VMs. In addition, NetHint provides to the tenant recent utilization summaries and counts of co-locating tenant connections on shared network links in the virtual topology. This information allows the tenant to adapt its transfer schedules based on both the topological and temporal hot spots in the network. Further, our design ensures that the only additional information NetHint exposes is aggregated network statistics across all tenants. It is thus difficult for a tenant to

acquire information about any individual other tenant. (§2.4.1)

For the second question, our preferred approach to collecting hints is to measure network traffic in the physical switches using network telemetry, e.g., sketching [171, 170]. However, sketches depend on specific programmable switch features, which are not widely deployed. Instead, our prototype employs a host-driven approach, in which each machine monitors local flows and transmits flow-level statistics to a NetHint measurement plane. One machine in each rack runs a *NetHint server* process to aggregate the rack-level information. These NetHint servers exchange information using periodic all-to-all communication. A cloud tenant connects to the local NetHint server to fetch hints. We show that this approach allows NetHint to provide timely hints to tenants with low CPU and bandwidth overheads (§2.4.2).

As for the third question, we consider two aspects of adaptation in response to the hints. First, we observe that the adaptation algorithm should take into account the application transfer schedule and semantics to maximize the performance gain. To this end, we consider several use cases for NetHint which cover a range of popular data-intensive applications, including (1) choosing allreduce algorithms in distributed deep learning, (2) constructing broadcast trees for serving ensemble models, and (3) placing tasks in MapReduce frameworks. For each case, we show how applications can adapt their transfer schedules based on the information in the hint. The takeaway is that for all of these examples, tenants can make use of the NetHint information via simple scheduling algorithms (§2.5).

Second, we explore the drawbacks of adaptation: it introduces extra computational overhead, and may be ineffective or even harmful or unstable if network conditions change too rapidly. We conclude that the adaptation algorithm should use different sets of hints depending on network changing frequency and adaptation overhead. For example, we find that if an application has a non-negligible latency to collect hints and compute the transfer schedules, the bandwidth information may be stale and thus may negatively affect the application performance (detailed in §2.6). Based on this intuition, we design a policy for applications to react to hints in a flexible manner: under stable network conditions and

Table 2.1: Notations and descriptions for NetHint.

Notations & Descriptions	
$\mathcal{T}$	A virtual topology connecting all the tenant's VMs
l	A virtual link in virtual topology $\mathcal{T}$
B_e^l	A tenant's bandwidth share on link l
B_t^l	Total bandwidth on link l
B_r^l	Residual bandwidth on link l
n^l	Number of shared objects on link l

low adaptation overheads, applications use both bandwidth and topology information to maximize the performance gain of adaptation. Otherwise, applications use only the stable topology information (§2.6).

2.4 Providing NetHint Service

2.4.1 What Is in the Hint?

NetHint exposes a virtual link-layer topology $\mathcal{T}$ to a cloud tenant. The tenant's virtual topology abstracts the network as a tree data structure in which the tenant's VMs are leaf nodes. A link in the tree represents one or more physical links in the data center network, and an interior node may abstract a region of switches and links. The prototype uses a three-layer tree that captures how VMs are distributed among racks in a data center and collapses the network structure above the rack level into a single root node. VMs residing in the same rack are in the same subtree. The virtual topology abstraction does not reveal racks or servers where the tenant has no presence. Following the common observation that congestion losses often occur at the rack level [295, 190, 129, 30], these virtual topologies in the NetHint prototype ignores congestion at any structure above the rack level [71]. It is possible to represent more structure by adding layers to the tree. The tree approximation presumes that the data center network is able to balance its load, so that traffic among children of an abstract node see similar available bandwidth. There is a rich literature on efficient network load balancing for data centers [6, 142, 292, 201, 69, 84, 106, 86, 63, 143], and some of them are readily deployable with commodity hardware.

NetHint allows applications to react to temporal hot spots in the network. For this purpose, NetHint exposes an estimate of utilization on each virtual link l. Recall Case 1 in Figure 2.2, now assume the orange flow from VM A uses only 2 out of 10 Gbps. If the tenant of VM B knows the network utilization information, it can infer that VM B can send traffic at 8 Gbps. As such, NetHint provides (1) the total bandwidth B_t^l and (2) the residual bandwidth B_r^l on each virtual link l. However, we find that this information alone is insufficient for an application to adapt its transfer schedule, especially when links are congested. For example, even if one link l has already reached 100% utilization, a tenant can still send flows through l and get a fair bandwidth share.

Shared objects and fairness models. In fact, the bandwidth share depends on the fairness model implemented by the cloud provider. Per-flow-fairness and per-VM-pair-fairness are enforced naturally for RDMA-based networks because modern RDMA NICs can be configured to choose either of them. Per-flow-fairness is ensured for containerized clouds because cloud users cannot modify the kernel TCP stack. For traditional TCP-based and VM-based clouds, many recent studies [107, 57, 199] describe how to enforce per-VM-pair-fairness. With the increasing programmability of modern switches, it now becomes possible to implement other fairness models in the network [240, 286], such as per-tenant fairness.

Consider an application placing 3 connections on a 100 Gbps network link with 7 existing connections from 3 other tenants. We assume each flow can reach 100 Gbps throughput. With per-flow fairness model, the application should get 30 Gbps bandwidth. With per-tenant fairness model, the application should get 25 Gbps bandwidth.

The example indicates that the bandwidth share also depends on the number of *shared objects* on each link l. The definition of shared object depends on the fairness model: it is a flow (VM-pair, tenant) under per-flow (per-VM-pair, per-tenant) fairness, respectively.

To provide bandwidth information, NetHint exposes the number of shared objects n^l on each link l. Taken together, NetHint provides a tuple (n^l, B_t^l, B_r^l), which includes both the current link utilization and the number of shared objects.

Bandwidth estimation. The information in the virtual topology enables a tenant to estimate its available bandwidth on each virtual link l efficiently. More formally, consider a tenant who plans to place k^l shared objects on link l in its transfer schedule. If link l is an in-network link in virtual topology $\mathcal{T}$ (i.e., not attached to any VM), the bandwidth share the tenant gets can be estimated as:

$$B_e^l = \max\left(\frac{k^l}{n^l + k^l} B_t^l, B_r^l\right) \tag{2.1}$$

Equation 2.1 indicates that when the link is under-utilized, the tenant can use up all the residual bandwidth B_r^l, and even if the link is already congested, the tenant can at least achieve its fair share based on the number of shared objects.

If link l is an edge link (i.e., attached to one VM), the bandwidth share is also affected by the underlying sharing approach. More specifically, denote the per-VM bandwidth guarantee provided by the sharing approaches as B_v, we have:

$$B_e^l = \begin{cases} \min(B_v, \max(\frac{k^l}{n^l+k^l} B_t^l, B_r^l)) & \text{static partitioning} \\ \max(\frac{k^l}{n^l+k^l} B_t^l, B_r^l, B_v) & \text{work-conserving} \end{cases} \tag{2.2}$$

Sources and impact of inaccuracy We acknowledge that both Equation 2.1 and Equation 2.2 are approximations and can sometimes be inaccurate. First, some shared objects (i.e., tenant, VM-pair, or connection) may have traffic demands less than their fair network share, thus calculating the exact value of B_e^l requires knowing the traffic demand for each shared object. NetHint does not provide per-object information, as doing so introduces security concerns and significant overhead given the huge number of such objects. Second, since a virtual link corresponds to the aggregation of multiple parallel paths in the physical topology, the estimation may be inaccurate under poor network load balancing across these parallel paths. We note that this is less likely to happen with recently proposed data center network load balancing designs.

Despite these inaccuracies in bandwidth estimation, our results (§3.6) show that even the three-level tree approximation is sufficient to adapt the transfer schedules and improve

the performance of our target applications. Moreover, evaluation results also show that the benefits degrade gracefully with the quality of the approximations.

Alleviating security and competitive issues. Compared with a naive white-box solution that exposes VM allocation information and physical network topology, NetHint has alleviated the security and the competitive concerns. First, NetHint does not expose the physical location of allocated VMs, so a tenant cannot learn the provider's VM allocation policy. Second, our network statistics are aggregated over all other tenants, so it is difficult for a tenant to infer from them the network behavior of any other individual tenant. Finally, network topology among a tenant's VMs is already accessible even in today's black-box model via user probing approaches, e.g., as presented in PLink [176] and Choreo [155]. NetHint does provide easier access to this information, but we believe this does not increase the security risks. Note that NetHint does not fully eliminate these issues, and we discuss them in §4.5.

2.4.2 Timely NetHint with Low Cost

User query overhead The virtual topology is presented as a set of links (each with a Link ID). Each virtual link has its associated B_t. The temporal utilization information for each link includes a tuple of three fields (Link ID, n, B_r). Each field occupies 8 bytes. As such, the amount of data returned by a query is small. Consider a cloud tenant that has rented 100 VMs allocated across 10 racks. As upstream and downstream virtual links are considered separately, the number of virtual links equals twice the sum of the number of VMs and the number of racks the tenant occupies. The amount of query information thus has $(100 + 10) \times 2 \times 3 \times 8 = 5280$ Bytes.

There is no value or incentive for a tenant to query at a higher frequency than the information update period of NetHint (100 ms by default). Tenant VMs communicate with a NetHint server through TCP connections with rate limits that prevent queries more frequent than once per 50 ms.

Collection overhead We design a two-layer host-driven aggregation approach to collect timely hint information with low cost. Recall that we select one machine in each rack to

run a NetHint server process. Each machine collects flow-level network characteristics from its operating system, and sends them to its rack-local NetHint server periodically. The information each machine has to send to the local NetHint server is a virtual link ID plus one $(n,\ B_r)$ for each virtual machine to ToR link and another $(n,\ B_r)$ containing only the traffic transmitting across the rack, for adding its contribution to the ToR uplink's $(n,\ B_r)$. Each field is 8 bytes, so the total data size per virtual link is $(1 + 2 \times 2) \times 8 = 40$ bytes. It is necessary to consider the upstream and downstream bandwidth independently, so each virtual machine or ToR has two associated virtual links. For example, assuming a physical machine has 10 VMs, it sends $40 \times 2 \times 10 = 800$ bytes of data to the NetHint server in each period. We set the information update period to 100 ms by default. Thus, the total aggregated information for one NetHint server is two $(n,\ B_r)$ for every VM-to-ToR virtual link and the ToR uplink in the virtual topology. The NetHint servers then use all-to-all communication to exchange their aggregated information.

Suppose a data center has 1000 racks, and every rack has 20 machines. In each information update period, a local NetHint server gathers 16 KB information (800 bytes $\times$ 20 machines). With a 100 ms update period, the total amount of cross-rack traffic introduced by the all-to-all information exchange is $16\,\text{MB}/100\,\text{ms} = 1.3\,\text{Gbps}$ per rack. Let's assume each rack has outgoing bandwidth of 500 Gbps. Then the bandwidth overhead of NetHint is 0.26%.

2.4.2.0.1 Failure detection and recovery NetHint is a best-effort service, and applications should be prepared to function without hints, e.g., if their rack-local NetHint servers become unavailable due to failures such as link failure and server crashing. In this case, applications just revert the transfer schedule to a default one assuming no known network characteristics until a new NetHint server is available in the rack.

2.5 Adapting Transfer Schedules with NetHint

We find that most data-intensive applications can be categorized into two classes, based on how they can adapt to network characteristics. For each application class, we show that

adapting transfer schedules corresponds to an optimization problem. Our goal here is not to present the optimal algorithm to solve the scheduling problems. Rather, our goal is to show that a broad set of distributed applications can benefit from NetHint using simple scheduling algorithms.

2.5.1 Optimizing Collective Communication

Many data-intensive applications run a high-level collective communication primitive (e.g., broadcast, allreduce) among a set of processes. Any such operation can be accomplished flexibly via a large set of possible *overlay topologies* among all the processes. For example, a broadcast can be performed with different broadcast trees connecting all the receivers, and an allreduce may employ different allreduce topologies (e.g., tree-allreduce or ring-allreduce). For all these communication primitives, the choice of overlay topologies affects only the efficiency (i.e., finish time) but not the correctness. Many popular ML applications belong to this category:

- **Data-parallel deep learning:** each server holds a replica of the model and calculates gradients locally. Servers use allreduce to synchronize gradients in each training iteration.
- **Reinforcement learning:** the trainer process in reinforcement learning repeatedly broadcasts the model (i.e., policy) to a dynamic set of agents.
- **Serving ensemble models:** multiple servers run DNN models simultaneously to predict the label on the same input data, and then use voting to decide the final output. For every input data batch, the front-end server broadcasts it to a set of servers holding different DNNs.

Moreover, as the object of collective communication is usually a vector of numbers, we can partition the object and apply different overlay topologies on each partition. For example, a broadcast can be accomplished via multiple broadcast trees, with each broadcast tree transferring a different (weighted) portion of the broadcast object. Similarly, an allreduce can be performed via a weighed combination of different allreduce topologies. The transfer schedule thus depends on both the choices of overlay topologies and their corresponding

weights.

With NetHint, the tenant can estimate the bandwidth B_e^l available on each link l based on Equation 2.1 and Equation 2.2. For a transfer schedule s, denote the volume it transfers on each link l as d_s^l. The corresponding latency of the schedule can be estimated as $\max_l(d_s^l/B_e^l)$. Thus, we have:

Problem statement: *Given the virtual topology $\mathcal{T}$ and the estimated bandwidth on each virtual link l, find a transfer schedule that minimizes the latency* $\max_l(d_s^l/B_e^l)$.

To solve the above problem, one major challenge is that the number of candidate transfer schedules can be huge. For example, there can be $O(n^{(n-2)})$ possible broadcast trees to broadcast a message to n processes [268]. One possible solution is to use tree packing algorithms [44, 268, 80]. However, since the goal here is to show the usefulness of NetHint information rather than to find the optimal algorithm, we design simple heuristics to solve the problem. We first sample a random set of overlay topologies (broadcast and allreduce trees) which cross each rack only once. We then use linear programming to find the best weight assignment among these trees, so that the transfer schedule minimizes the latency $\max_l(d_s^l/B_e^l)$.

2.5.2 Optimizing Task Placement

Many distributed applications execute based on a task graph describing the tasks and their dependencies. The task graph can be static (i.e., task graph is known before the workload runs) [60, 289] or dynamic (i.e, tasks arrive as the workload runs) [193]. Since different tasks may send and receive different amounts of data, the placement of tasks onto VMs determines the transfer schedule among the VMs. Applications in data analytics frameworks and task-based distributed systems therefore can benefit from network-aware task placement:

- **Data analytics frameworks** [289, 100]: data analytics workloads contain network-intensive shuffle phases between different job stages. One shuffle phase creates an all-to-all communication between a set of sender tasks and receiver tasks, so task placement controls the

Table 2.2: Important factors related to the impact of staleness.

Notations & Descriptions	
T_b	Average changing period of the background network condition
T_u	Duration of a transfer schedule being used
T_a	Latency to adapt (collecting information and computing a schedule)
T_s	Staleness of the hint
p	A threshold defined by the ratio between total adapting latency and JCT

shuffle performance.

- **Task-based distributed systems** [193, 113] are increasingly popular in industry. In these applications, the task graph is dynamic and generated at runtime. Tasks launch after fetching input objects from upstream tasks. As such, efficient task placement can minimize the task launch latency reducing the object fetch time.

Problem formulation For both applications, we can formulate the task placement as a classical network embedding problem. Denote the set of tasks as $\mathbb{T}$ and the set of VMs as $\mathbb{V}$. Compared with the problem statement in §2.5.1, which selects an efficient data transfer schedule, here we need to find an embedding $\mathcal{E} : \mathbb{T} \mapsto \mathbb{V}$ given the transfer schedule among all tasks. The algorithm inputs and optimization goals are the same as the problem statement in §2.5.1, except that the latency is calculated as $\max_l(d_e^l / B_e^l)$. d_e^l is the transfer volume on link l introduced by embedding $\mathcal{E}$.

We make minor modifications to the greedy heuristics proposed in Hedera [5] to solve the embedding problem. We first sort all tasks in $\mathbb{T}$ based on the amount of data they receive in decreasing order (no need if $|\mathbb{T}| = 1$). We then place tasks one by one following this order. When placing a task to $\mathbb{V}$, we optimize greedily for the objectives described in the problem statement. Before processing the next task, we update the cross rack traffic and d_e^l based on the placement.

2.6 Flexible Adaptation for Stale Information

Staleness of NetHint information The staleness of NetHint information during job execution is affected by the following two factors (notations listed in Table 2.2). First, an application controller can have a non-negligible latency to collect hints and compute the transfer schedules based on the hints, which makes the hints stale when being applied. We denote the adaptation latency as T_a.

Second, applications can adapt to hints periodically. For each adaptation period, the schedule calculated based on the previous hint will be used for the entire duration T_u. Note that for recursive jobs (e.g., model serving), recomputing the schedule for every iteration introduces too much latency. To this end, we fetch hints and recompute the schedule every k iterations, so that the latency to compute transfer schedule is within a portion p (e.g., 10% by default) of the job execution time. Moreover, for jobs that adapt the task placement based on hints (e.g., MapReduce), the adaptation period T_u equals job completion time, as the task placement usually cannot be changed during job execution.

Taken together, the staleness of NetHint information is quantified as $T_s = T_a + T_u$, which is the combination of both above factors. T_a is the total latency of four steps. The first three steps are to collect hints: sending host network characteristics to NetHint service, NetHint service exchanges rack-level network characteristics, and applications querying the NetHint service. The maximum latency for these three steps combined is 300 ms (100 ms per step due to NetHint frequency), so we use 150 ms as the estimate for the average case latency. The last step is to compute the transfer schedule, and it is application-specific (Figure 2.7). In our evaluation, a deep learning job of 64 workers requires 10 ms to compute its transfer schedule. We thus set $T_a = 150 + 10 = 160$ ms. We set $T_u = 100$ ms to keep the compute overhead to be less than 10% of the total running time.

Impact of the stale information The impact of stale information depends on the relative relationship between (1) the staleness of the information; and (2) the stability of the underlying network condition. Assume the background network condition changes every T_b time

in average. A hint with staleness T_s much less than T_b can still be helpful since the network condition is likely to be similar with the condition T_s time ago. In contrast, a hint with staleness T_s much larger than T_b will be misleading, since the current network condition may be very different from the condition T_s time ago. In this case, adaptation with misleading hints can negatively affect the application performance (Figure 2.12d).

Flexible adaptation based on application and network condition. There are two takeaways from the above analysis. First, stale information should not be used when it is misleading. Regarding this, one approach is to simply ignore the provided hints and run applications as we run them today. However, as we show in motivating examples (e.g., Figure 2.1c and Figure 2.4c), the link-layer network topology alone can be useful for some types of applications to reduce the amount of cross-rack traffic. Compared with the bandwidth information, topology information is more stable and not affected by network dynamics.

Therefore, we propose NetHint-TO, a class of scheduling algorithms that use only the stable topology information from NetHint. For example, with NetHint-TO, we create a ring that crosses each rack only once for ring-allreduce and a chain that crosses each rack only once for tree-broadcast.

The second takeaway is that there is no one-size-fits-all solution. Each application should have two scheduling algorithms, one uses bandwidth information (in §2.5) and another one uses stable topology information only (NetHint-TO). We design a policy to choose between these two algorithms based on both the application and the network conditions (i.e., T_b, T_u, T_a). More specifically, when $T_s < T_b$, applications use the scheduling algorithm in §2.5 to calculate the optimal schedule based on both bandwidth and topology information. When $T_s \geqslant T_b$, applications adopt NetHint-TO to minimize the impact of stale information.

2.7 Implementation

We implement NetHint using 4600 lines of Rust code. 2300 additional lines of code are in NetHint server to provide NetHint to cloud tenants. The algorithms for applications to adapt transfer schedules (i.e., MapReduce, allreduce, and broadcast) are implemented using

149, 216, and 144 lines of code. We use `lpsolve` [174] for solving linear programs.

To compute the hints in our testbed, we take an endhost-based approach. We hook an eBPF program into the OS kernel. The eBPF program counts the total number of bytes going within the rack and outside the rack. A userspace program polls the counters from the eBPF program every 10 ms and maintains a moving average of the number of existing shared objects (i.e., flows, in a per-flow fairness model). The userspace program sends the number of shared objects and traffic data to the NetHint server every 100 ms. In a deployment environment where SmartNICs is available, we can also program the SmartNICs to implement this logic.

NetHint server binds to a TCP port, where VMs connect to to fetch hints. NetHint server uses a single thread to respond to NetHint queries. A single thread is enough for our design because queries are not frequent.

For an application to use NetHint, we need to modify the application. For traditional collective communication, the transfer schedule is static and decided before runtime. Recent collective communication designs have shown that transfer schedules can be dynamically decided at runtime [306]. NetHint can help these dynamic collective communication designs to decide on an efficient transfer schedule based on network characteristics. These dynamic collective communication designs can query and adapt transfer schedule every k iterations before issuing data transfer operation. For task placement, the global scheduler of a distributed system (e.g., master in MapReduce [60]) queries the NetHint server and uses both the task information and the NetHint information to decide task placement. For our evaluation purpose, we build a dynamic scheduler for collective communication and a task scheduler for MapReduce tasks according to the descriptions above.

2.8 Evaluation

2.8.1 Setup and Workloads

We evaluate NetHint using an on-premise testbed and large-scale simulations. Our setting is that hosts ensure work-conserving bandwidth guarantee for VMs and the network

ensures per-flow fairness. We compare NetHint with the scenarios where cloud tenants (1) do not consider network characteristics and (2) probe the network to reverse-engineer the network characteristics and then adapt transfer schedules. For user probing, we assume network information is always correctly reverse engineered. We assume the probing strategy is the following: For a tenant that owns n hosts, user probing runs in $n/2$ rounds, where each round's latency is either the latency to send 10000 packets or 1 second, whichever is smaller, to measure throughput and latency between $n/2$ pairs of hosts. [2] Similar to NetHint, user probing adopts the same strategy to periodically update the transfer schedule, but with a lower frequency due to its higher overheads. We calculate user probing's frequency using the same method described in the second paragraph of §2.6.

We use a mix of two types of background traffic to simulate skewed and long-tailed traffic in data centers [295, 232, 30, 7]. One slow-moving background traffic occupies 0-50% bandwidth of the link capacity on each link in a Zipfian distribution. The slow-moving background traffic occupies 10% bandwidth in total and changes every 10 seconds. The other is a fast-moving background traffic which is on all links and occupies 0-10% bandwidth of the link capacity in a uniform random fashion. The fast changing background traffic changes every 10 ms. We use the following workloads. We run each experiment 5 times and report the average speedup for each job. To quantify the overall speedup, we also measure the arithmetic average of speedups across jobs.

Distributed data-parallel deep learning. We test the allreduce completion time. The job sizes are either 16 or 32 (in terms of number of nodes) with equal probability. For each allreduce job, we set the buffer size to be 100 MB ($\approx$ the size of ResNet-50). We run 100 jobs and assume jobs arrive as a Poisson process. We choose Poisson lambda = 24 seconds, so that the average network utilization approximates to 12%.

Serving an ensemble of ML models. We test the broadcast completion time. We use

[2] We believe this is a best-case scenario for existing user probing techniques. Plink [176] sends 10000 packets per VM-pair to reverse engineer link-layer topologies. Choreo [155] uses a 3-step strategy to pinpoint congested links and its first step is measure pair-wise bandwidth. It takes 3 minutes to reverse engineer the network conditions for 10 VMs (90 VM-pairs).

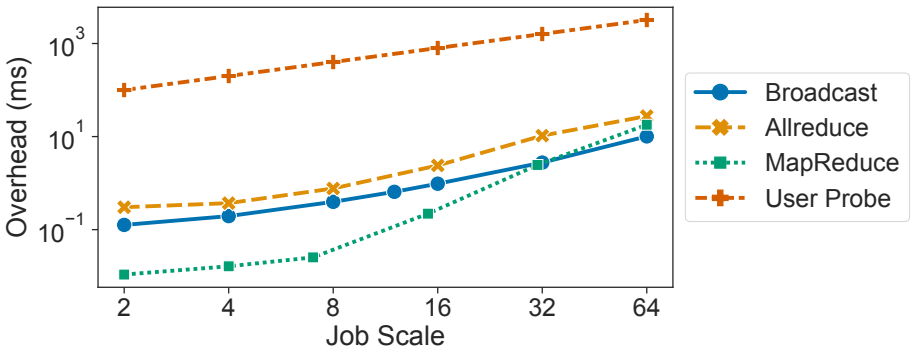


FIGURE 2.7: **Testbed results:** Latency to compute transfer schedules.

the same job size distribution described in Hoplite [306]. We run 100 jobs and assume jobs arrive as a Poisson process. We choose Poisson lambda = 8 seconds, so that the average network utilization approximates to 12%.

MapReduce. We test the latency of the data shuffling phase of MapReduce. We use Facebook's MapReduce trace [54], which contains 500 MapReduce jobs and their arrival time. We assume the traffic is divided evenly from a reducer to the mappers.

2.8.2 NetHint in Testbed Experiments

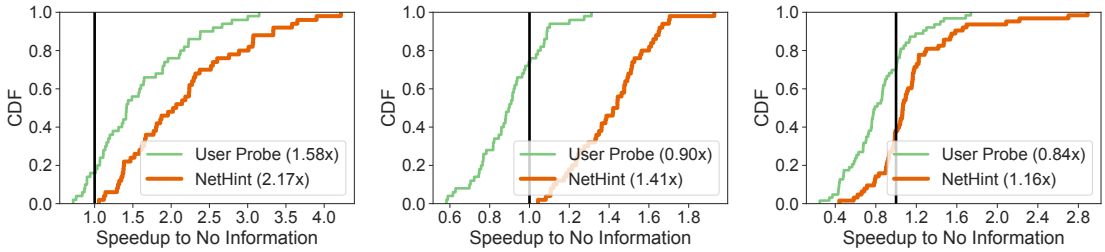
We build a 6-server testbed. Each server has a 100 Gbps Mellanox ConnectX-5 NIC and two Intel 10-core Xeon Gold 5215 CPUs (2.5 GHz). These machines are connected via an emulated 40 Gbps 2-stage FatTree network using a single 100 Gbps Mellanox SN2100 switch through self-wiring. 3 machines are in one rack, and the rest 3 machines are in the other rack. The oversubscription ratio on our network is 3. Each machine runs 4 VMs where each VM is guaranteed 10 Gbps through fair-queuing on the NICs.

Overheads. We already provide analysis of bandwidth overheads in §2.4.2. Now the remaining question is how much overhead NetHint incurs in terms of latency and CPU cycles. Collecting statistics from eBPF program is instant, and the polling period for flow statistics is 10 ms.

To measure the overheads in large deployment, we use each CPU core in our testbed to emulate a rack by instantiating a NetHint server per-core. We use `pidstat` to measure the

Table 2.3: **Testbed results:** The system overhead of a NetHint server in CPU utilization, memory, and information collection latency.

# Racks	CPU Util. (%)	Memory (MB)	Latency (ms)
6	0.06	4.53	10.60
24	0.14	5.90	10.73
96	0.41	19.28	11.91
240	0.66	78.16	13.73



(a) Distributed deep learning (b) Ensemble model serving (c) MapReduce

FIGURE 2.8: **Testbed results:** NetHint's speedup on testbed for allreduce in data-parallel distributed training, broadcast in ensemble ML model serving, and mapreduce shuffle compared with user probing and not using network information. Numbers in the legend shows the average of speedups compared with running applications without network information.

CPU cycles and memory footprint on NetHint server. Table 2.3 shows the result. When the number of racks scale up to 240 racks, the CPU time spent on NetHint servers is negligible, i.e., less than 0.66%. The memory footprint on each NetHint server is small (less than 80 MB) and scales with the number of racks mainly due to the increase in the hint size. The latency to collect network information is less than 14 ms.

We implement the algorithms described in §2.5. We test the computation latency of running each algorithm at different scales (number of workers). Figure 2.7 presents the results. The latency to make a scheduling decision remains low, ranging from 10 us to 30 ms. Compared with the computation latency, the extra latency introduced by user probing is much higher, ranging from 100 ms to 3 seconds. The round-trip latency to fetch hints takes 100 us because it is rack-local.

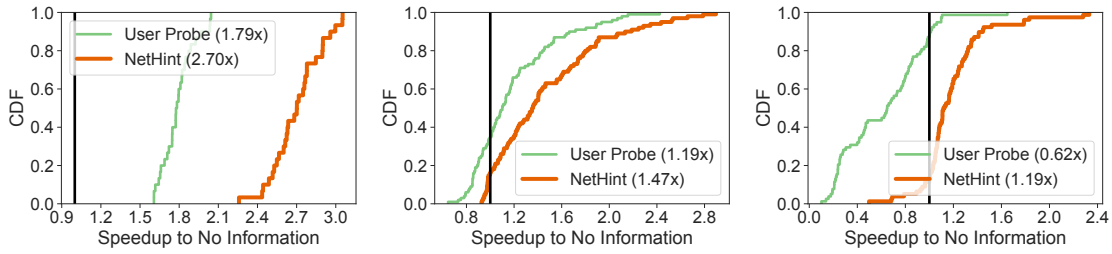
Results. NetHint improves application performance. Figure 2.8 shows the normalized

speedup to running applications without network information. Using user probing speeds up the communication by 1.6x for distributed data-parallel deep learning and slows down the communication by 1.1x and 1.2x for serving an ensemble of ML models, and MapReduce shuffle, respectively. NetHint speeds up communication of these workloads by 2.2x, 1.4x, and 1.2x, substantially outperforming user probing. NetHint can outperform user probing because collecting hints is more lightweight than each application individually probing the network characteristics. User probing hurts many ensemble model serving and MapReduce jobs because of the probing overheads. In addition, we notice that a small portion of jobs in Figure 2.8c are penalized. On our testbed, the job log shows that there are on average 2.8 jobs sharing the rack bandwidth. One job arrival or departure changes the network condition for all the other jobs on the rack. However, the task placement decision cannot be changed during job execution, and thus the initial placement can be imperfect. In contrast, deep learning and model serving workloads in Figure 2.8 do not severely suffer from this problem, as they can timely modify the transfer schedule for each iteration based on the latest NetHint information.

2.8.3 NetHint in Simulations

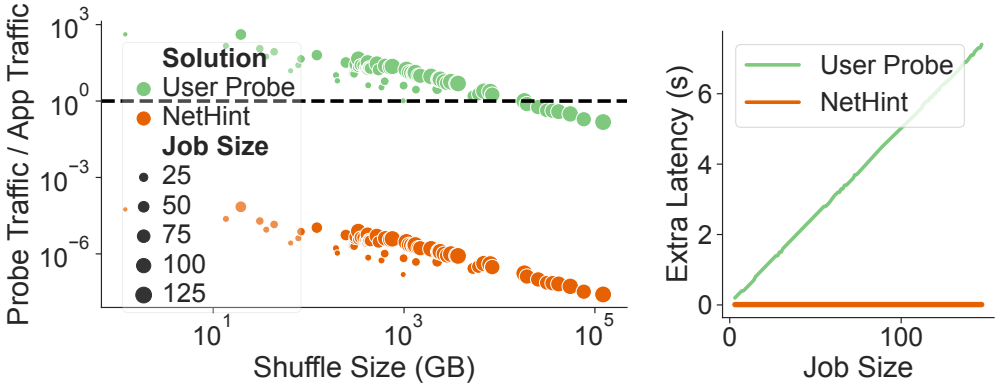
We use simulations to evaluate NetHint in large-scale deployments and in various operating environments. Our simulator is written in 5000 lines of Rust. The simulation is at flow level, and throughput of each flow is the result of solving a max-min fairness formula based on traffic demand. We simulate a CPU cluster and a GPU cluster individually. Both the CPU and GPU clusters have 150 racks. In the GPU cluster network, each rack has 6 machines with 100 Gbps NIC, and each rack has total upstream bandwidth of 200 Gbps. In the CPU cluster network, each rack has 18 machines with 100 Gbps NIC and the total upstream bandwidth is 600 Gbps. The oversubscription ratios are both 3. In the CPU cluster, each machine has 4 VMs. In the GPU cluster, each machine only has 1 VM. All VMs have bandwidth guarantee of 25 Gbps.

Results. Figure 2.9 shows the NetHint's speedup of the three workloads in our simula-



(a) Distributed deep learning (b) Ensemble model serving (c) MapReduce

FIGURE 2.9: **Simulation results:** Comparing NetHint with dynamic user probe in the default background traffic setting.

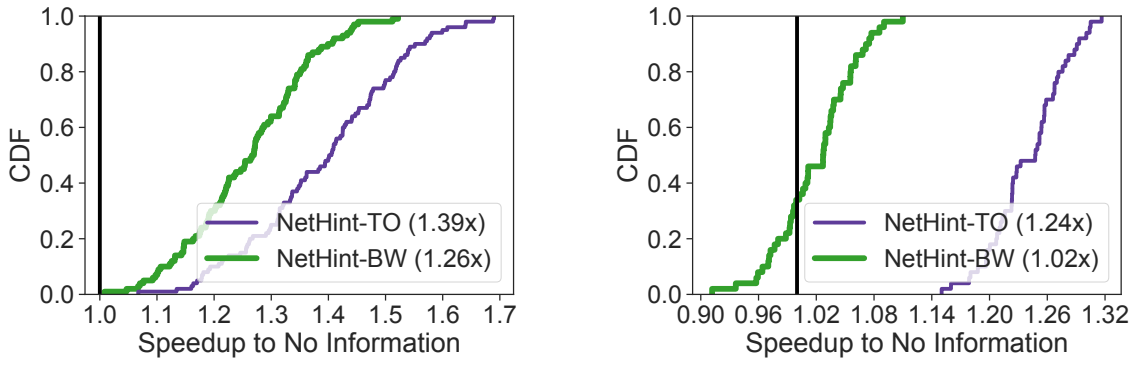


(a) Probing traffic over application traffic (b) Extra time

FIGURE 2.10: **Simulation results [MapReduce]:** Extra overhead for MapReduce jobs comparing NetHint and user probing.

tions. In summary, the trend of the simulation results matches what we have observed on the testbed. NetHint speeds up communication by 2.7x, 1.5x, and 1.2x, respectively. On allreduce, the speedup is higher than that on the testbed because the number of hosts involved in a job is larger than that on the testbed, and thus the amount of cross-rack traffic is also larger, giving NetHint more room to optimize transfer schedules.

User probing incurs substantial overheads in both traffic and latency. Figure 2.10 shows the overheads of using NetHint and user probing in MapReduce. The amount of overhead depends on both MapReduce shuffle size and job size. Figure 2.10a shows the extra traffic introduced by NetHint and user probing over application traffic. NetHint only adds less than 0.1% extra traffic. User probing, in contrast, adds 15% to 420% extra traffic, and 90%



(a) Coarse-grained workloads

(b) Non-negligible overhead to compute a transfer schedule

FIGURE 2.11: **Simulation results [Model serving]:** Using topology information alone can outperform using bandwidth information.

of jobs double their traffic. This is because user probing needs to generate probe traffic, and each application has to probe independently. For large shuffle sizes, the probing traffic is less of a concern because it constitutes a smaller fraction of the total traffic. Figure 2.10b shows the extra latency due to probing and fetching hints for MapReduce jobs of various sizes. NetHint only adds a constant RTT-level extra latency which is negligible. User probing has a large latency overhead, which is linear in job size. This is expected because user probing needs to run for $n/2$ rounds, where n is the job size. There are a set of MapReduce jobs that are penalized substantially by user probing (as shown in Figure 2.9c). These are MapReduce jobs with large job sizes but with small shuffle sizes.

When should NetHint use topology information only? As we have described in §2.6, there are two situations we prefer letting NetHint use topology information only: (1) workload granularity is large, and (2) overhead of computing a transfer schedule is non-negligible. To demonstrate these situations, we set the slow-moving background traffic change frequency to every 0.2 seconds. Other environment settings remain the same as those in previous simulations.

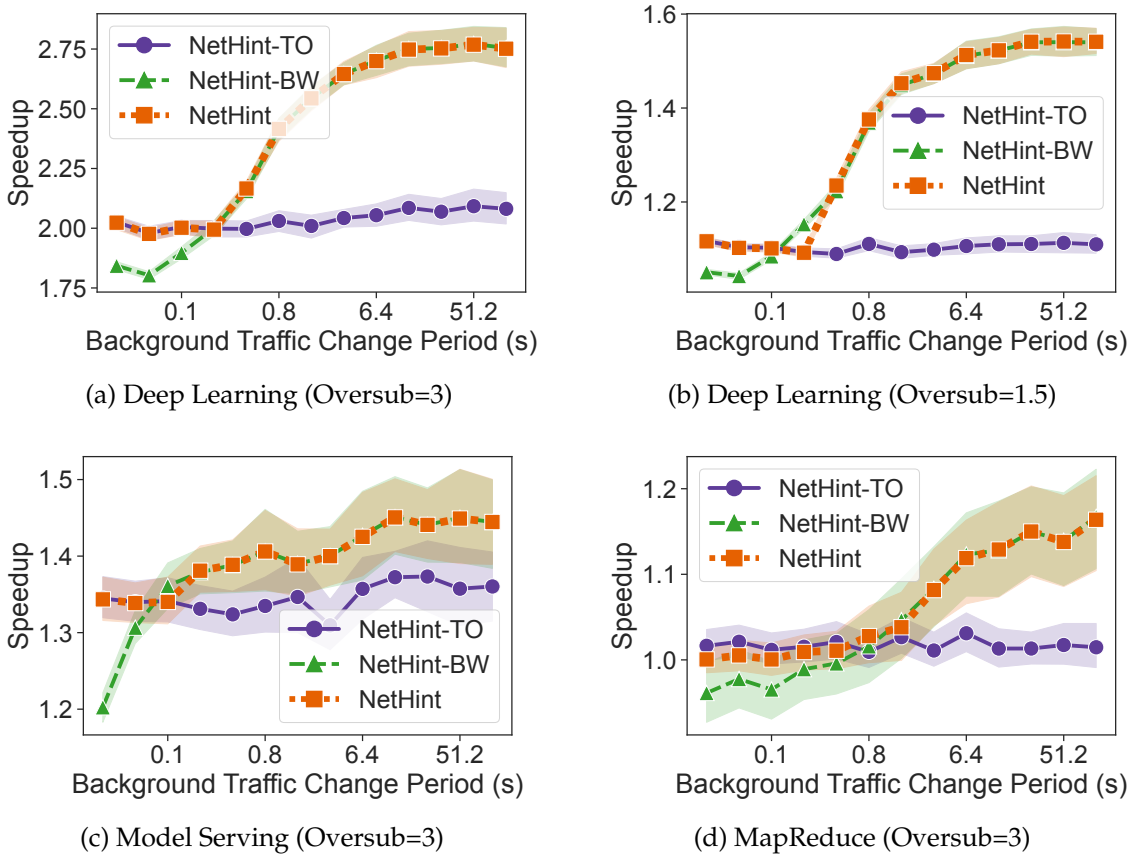
To show the case when background traffic changes faster than job completion time, we run 100 broadcast jobs with the model sizes increased to 1 GB. We let NetHint recompute

a new broadcast strategy every iteration (but we still guarantee that the computational overhead is under a certain threshold $p = 10\%$). We use NetHint-TO to denote using only topology information when calculating the transfer schedule. We use NetHint-BW to denote using bandwidth information when calculating the transfer schedule. Figure 2.11a shows that NetHint-TO and NetHint-BW speed up the communication by 1.4x and 1.3x. NetHint-BW is slightly slower than NetHint-TO. Applying a bandwidth-aware algorithm does not bring benefit compared with using topology information only because the background traffic changes even within a single broadcast. Instead, it can slow down the job due to the additional overhead to compute data transfer schedules.

To demonstrate an extreme example for the computational overhead, we run 100 broadcasts of 64 workers with data size set to 12 MB, and we double the bandwidth capacity of ToR switch. Figure 2.11b shows that NetHint-TO and NetHint-BW speed up by 1.2x and 1.0x compared with no information. NetHint-BW cannot improve because the computation latency using LP is large in contrast to the broadcast latency on such a small data size. It has to adapt its traffic less frequently ($\approx 0.2\,s$) to ensure the compute overhead is within 10% of the total job completion time. Without being affected by inaccurate hints, NetHint-TO aims to minimize the cross-rack traffic, thus achieving better performance.

Figure 2.12 shows which adaptation method NetHint choose under different background traffic change periods and oversubscription ratios. The result demonstrates that NetHint chooses the best of NetHint-TO and NetHint-BW for all the three applications we use and also for both oversubscription ratio of 3 and 1.5.

Inaccurate bandwidth estimation. The bandwidth estimations in Equation 2.1 and Equation 2.2 is based on approximations, as the accurate estimation requires knowing the traffic demand for each tenant. One question to ask is whether NetHint's design fundamentally relies on the accuracy of bandwidth estimation. To answer this question, we intentionally add noise to the input of NetHint. Having additional noise of x% means the link utilization provided to NetHint is between 100-x% and 100+x% of the actual utilization. We then evaluate the speedup of allreduce and broadcast jobs. Figure 2.13 shows the result. NetHint's



(a) Deep Learning (Oversub=3)

(b) Deep Learning (Oversub=1.5)

(c) Model Serving (Oversub=3)

(d) MapReduce (Oversub=3)

FIGURE 2.12: **Simulation results:** Average speedup to background traffic change period under two different topology settings. The shaded area represents 95% confidence interval.

speed up degrades gracefully. NetHint can still outperform not using network information when there is up to at most 50% noise.

Performance stability. To evaluate if NetHint's performance remains stable when the number of NetHint users is large, we increase the number of overlapped jobs. For deep learning, we enlarge the rack size to allow more jobs to share a ToR link and start all the jobs at the beginning. For MapReduce, we scale up the job arrival rate to create more overlapping among jobs. Figure 2.14 shows that NetHint can constantly achieve performance gain over not using network information.

Sensitivity to network configurations. We evaluate NetHint's speedup under different network configurations in terms of the number of machines per rack and oversubscription

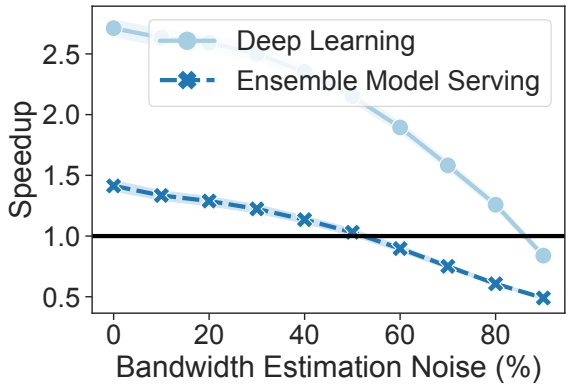


FIGURE 2.13: **Simulation results:** NetHint's speedup to not using network information when we add noise to the input of NetHint.

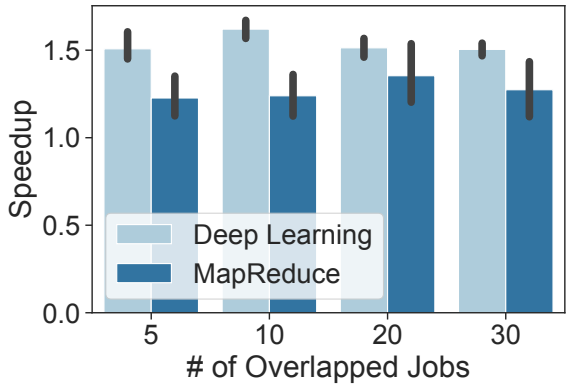
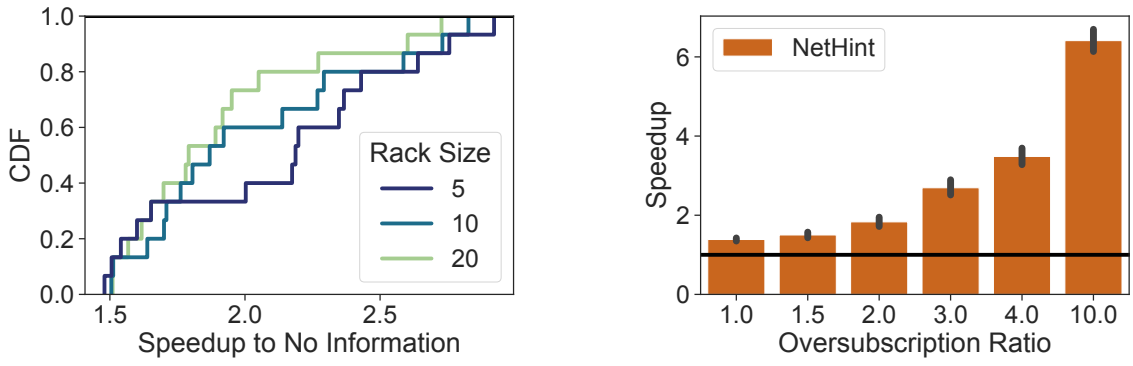


FIGURE 2.14: **Simulation results:** NetHint's performance when varying the number of overlapped jobs.

ratios. We vary the number of machines per rack while keeping the oversubsription the same at 3. Figure 2.15a shows that NetHint can reduce the communication latency consistently for different rack sizes. We then vary the oversubscription ratio. Figure 2.15b shows that NetHint's improvement compared with not using network information increases as oversubscription ratio increases. This is because, when oversubscription ratio is high, the cross-rack communication is more likely to become the bottleneck. NetHint can mitigate this bottleneck by reducing the total amount of cross-rack traffic.

Performance gain over perfect user probing. In our evaluation, for n hosts, user probing is performed in $n/2$ rounds. In each round, it measures the bidirectional bandwidth and



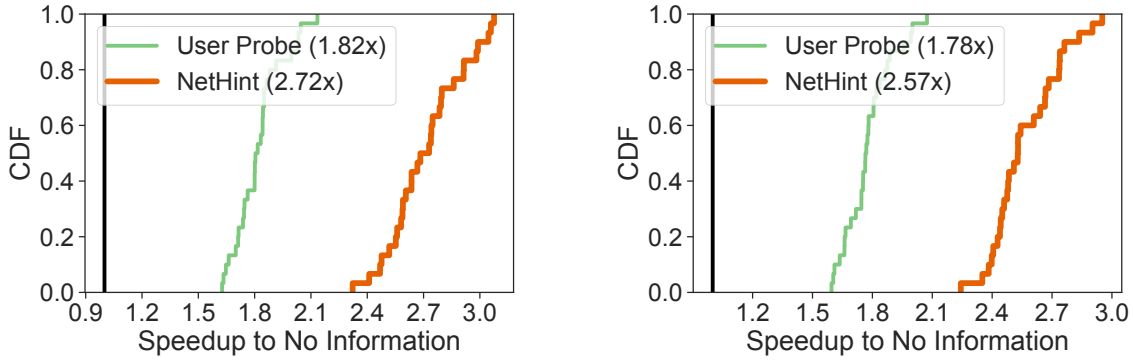
(a) Number of machines per rack

(b) Oversubscription ratios

FIGURE 2.15: **Simulation results [Distributed deep learning]:** NetHint's speedup to not using network information under different deployment environments.

latency between $n/2$ pairs of hosts in parallel for a certain duration (default to 100 ms). Moreover, we show some evidence that it can be difficult to design better user probing technique to achieve similar performance as NetHint. First, we demonstrate how low the user probing duration has to be in order to achieve similar performance as NetHint. For this, we artificially reduce the probing duration while ensuring probing is accurate in simulations. Figure 2.17a shows the result: even when probing duration is reduced to 1 ms, NetHint still has a small performance advantage over user probing. Second, we show that such a low probing duration (i.e., 1 ms) for accurate bandwidth estimation can be difficult due to data center microbursts. We simulate data center microbursts based on measurement results in Facebook data centers [295] and calculate whether probing for x ms is sufficient to predict the average bandwidth of 100 ms. Figure 2.17b shows that if we measure for less than 25 ms, there is a 50% probability that the estimation error is above 75%. This is because there are gaps between microbursts, when a busy link is temporarily idle. Probing for such a short amount of time may not detect any traffic.

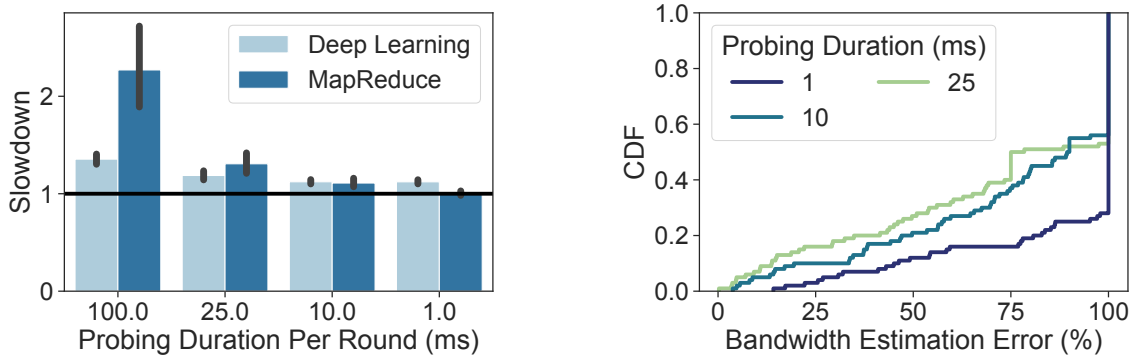
Does NetHint work for other fairness models? The rapid advancement in the programmability in emerging programmable switches makes it possible to implement other types of fairness models in the network [240, 286]. This trend makes it interesting to also understand NetHint's potential performance gains if we move to other fairness models in the



(a) Per-tenant fairness

(b) Per-VM-pair fairness

FIGURE 2.16: **Simulation results [Distributed deep learning]:** Speedup for other fairness models.



(a) Sensitivity to probing cost

(b) Probing accuracy

FIGURE 2.17: **Simulation results:** The speedup of user probing to NetHint and the relative bandwidth estimation difference under different assumptions of probing durations. The black line in (a) represents NetHint.

future. We simulate the same allreduce jobs except that we modify our simulator for different fairness models. As shown in Figure 2.16, the trend of the simulation results matches what we have obtained in a per-flow based fairness setting.

2.9 Discussion

Herd behaviors. Tenants adapting transfer schedules with provided hints in a distributed way can potentially cause stability issues. For example, given the information of an underutilized link, many tenants may make identical choices to move traffic to this link, causing

congestion. Such herd behavior causes load imbalance and performance oscillation in distributed load balancing problems [6, 292, 186]. We note that herd behavior is a common problem in some specific applications such as distributed load balancers. There are also standard techniques such as adding random jitters, and power of two choices to alleviated herd effect [186]. Whether and how NetHint should help specific applications avoid herd behavior is an interesting future direction. In the workload and setting of our evaluation, NetHint's speedup does not decrease when we increase the number of overlapped jobs (Figure 2.14). This infers that the performance of NetHint is not significantly affected by herd behavior.

Other competitive concerns for NetHint. NetHint exposes network utilization information to tenants. Network utilization can be a sensitive information. For example, one can infer whether a cloud provider suffers from in-network congestion. NetHint makes it easy for a customer to compare network characteristics at different times. If a customer finds that the achievable bandwidth is reducing via NetHint, there may be a risk that the customer will switch to another cloud provider.

2.10 Related Work

Sharing network bandwidth. How to share network among many applications or cloud tenants is one of the oldest problems in computer networks. Today, network sharing is opaque to the application or cloud tenants. Within a single tenant, bandwidth sharing is through the fairness property of the underlying congestion control algorithms [78]. Across tenants, a cloud provider usually enforces strong isolation through static bandwidth allocation [226] or work-conserving bandwidth guarantee [22, 156, 23] on the NICs. It is difficult to enable either static bandwidth allocation or work-conserving bandwidth guarantee in the network because commodity switches have limited numbers of hardware queues. NetHint is complementary to these bandwidth sharing design: NetHint does not change any fairness property of the network. NetHint provides guidance for applications to use the network bandwidth better. A non-participating tenant can simply ignore the hint.

Collective communication and task placement based on network characteristics. Many related works optimize collective communication [209, 140, 268, 61, 89] or task placement [119, 299, 155, 241, 270] based on topology or bandwidth information. Similar considerations can also be applied inside OS for multi-core machines [26]. Most of these solutions assume the network topology or bandwidth information is already known. As such, NetHint can work in complementary with these solutions by providing them timely network information. Second, these works do not consider a multi-tenant environment. They assume workloads can be controlled by a logically centralized controller, while we assume each tenant's workload is controlled only by the tenant itself. Because tenants do not know other tenants' communication patterns, this knowledge needs to be provided either through cloud provider's support as proposed in this work or using probing.

User probing. In addition to PLink and Choreo, many past works [279, 9, 237] also propose to measure network characteristics in wide-area networks to choose Internet route. NetHint is different in two aspects: (1) NetHint does not rely on active probe, and thus NetHint has low cost. NetHint simply reads counters directly from NICs or operating systems. (2) NetHint is for distributed applications that can adapt their transfer schedules rather than choosing routes in the network.

2.11 Summary

Today, the networking abstraction a cloud tenant has is a black box. This prevents a tenant's data-intensive applications from adapting the data transfer schedules to achieve high performance. We design and implement NetHint, a new paradigm for division of work between a cloud provider and its tenants. A cloud provider provides a hint, network characteristics (e.g., a virtual link-layer network topology, number of co-locating tenants, available bandwidth), directly to its tenants. Applications then adapt their transfer schedules based on these hints. We demonstrate the performance gain of NetHint on three use cases of NetHint including allreduce communication in distributed deep learning, broadcast in serving ensemble models, and scheduling tasks in MapReduce frameworks. Our evaluations show

that NetHint improves the performance of these workloads by up to 2.7×, 1.5×, and 1.2×, respectively. Our source code is available at `https://github.com/crazyboycjr/nethint`.

3. Remote Procedure Call as a Managed System Service

Having addressed the challenge of opaque cloud networking in chapter 2 with the NetHint system, we now turn to a different, but complementary, problem in application-network co-design. NetHint showed how exposing network hints can significantly improve data-intensive application performance by enabling cross-layer optimizations. In the next chapter, we focus on the communication patterns of cloud microservices and the inefficiencies in the current RPC (Remote Procedure Call) management model. While chapter 2 broke the network's black-box to aid application scheduling, chapter 3 will elevate RPC handling into the operating system to streamline how services communicate. This transition—from network visibility to communication management—continues our overarching theme: redesigning interfaces and services at the boundary of applications and networks. By moving upward in the software stack, chapter 3 introduces mRPC, a solution that centralizes and optimizes RPC processing to eliminate the overheads.

3.1 Introduction

Remote Procedure Call (RPC) is a fundamental building block of distributed systems in modern datacenters. RPC allows developers to build networked applications using a simple and familiar programming model [36], supported by several popular libraries such as gRPC [93], Thrift [247], and eRPC [135]. The RPC model has been widely adopted in distributed data stores [137, 243, 70], network file systems [235, 88], consensus protocols [202], data-analytic frameworks [60, 242, 288, 42, 262, 12, 173, 90], cluster schedulers and orchestrators [152, 109], and machine learning systems [208, 2, 194]. Google found that roughly 10% of its datacenter CPU cycles are spent just executing gRPC library code [138]. Because of its importance, improving RPC performance has long been a major topic of research [36, 239, 33, 34, 266, 252, 264, 135, 188, 161, 50].

Recently, application and network operations teams have found a need for rapid and flexible visibility and control over the flow of RPCs in datacenters. This includes monitoring and control of the performance of specific types of RPCs [187], prioritization and rate

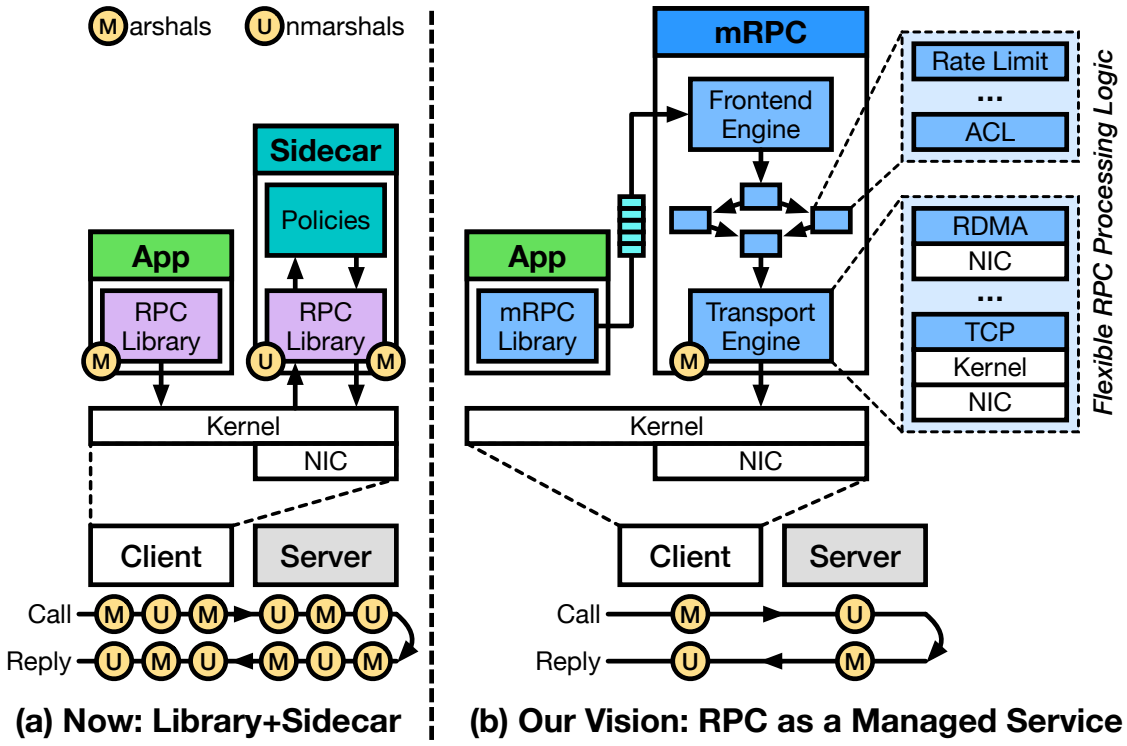


FIGURE 3.1: Architectural comparison between current (RPC library + sidecar) and our proposed (RPC as a managed service) approaches.

limiting to meet application-specific performance and availability goals, dynamic insertion of advanced diagnostics to track user requests across a network of microservices [77], and application-specific load balancing to improve cache effectiveness [32], to name a few.

The typical architecture is to enforce policies in a sidecar—a separate process that mediates the network traffic of the application RPC library (Figure 3.1a). This is often referred to as a service mesh. A number of commercial products have been developed to meet the need for sidecar RPC proxies, such as Envoy [68], Istio [115], HAProxy [103], Linkerd [167], Nginx [198], and Consul [55]. Although some policies could theoretically be supported by a feature-rich RPC runtime linked in with each application, that can slow deployment—Facebook recently reported that it can take *months* to fully roll out changes to one of its application communication libraries [75]. One use case that requires rapid deployment is to respond to a new application security threat, or to diagnose and fix a critical user-visible

failure. Finally, many policies are mandatory rather than discretionary—the network operations team may not be able to trust the library code linked into an application. Example mandatory security policies include access control, authentication/encryption [55], and prevention of known exploits in widely used network protocols such as RDMA [231].

Although using a sidecar for policy management is functional and secure, it is also inefficient. The application RPC library marshals RPC parameters at runtime into a buffer according to the type information provided by the programmer. This buffer is sent through the operating system network stack and then forwarded back up to the sidecar, which typically needs to parse and unwrap the network, virtualization, and RPC headers, often looking inside the packet payload to correctly enforce the desired policy. It then re-marshals the data for transport. Direct application-level access to network hardware such as RDMA or DPDK offers high performance but precludes sidecar policy control. Similarly, network interface cards are increasingly sophisticated, but it is hard for applications or sidecars to take advantage of those new features, because marshalling is done too high up in the network stack. Any change to the marshalling code requires recompiling and rebooting each application and/or the sidecar, hurting end-to-end availability. In short, existing solutions can provide good performance, or flexible and enforceable policy control, but not both.

In this chapter, we propose a new approach, called RPC as a managed service, to address these limitations. Instead of separating marshalling and policy enforcement across different domains, we combine them into a single privilege and trusted system service (Figure 3.1b) so that marshalling is done **after** policy processing. In our prototype, mRPC for managed RPC, the privileged RPC service runs at user level communicating with the application through shared memory regions [34, 27, 179]. However, mRPC could also be integrated directly into the operating system kernel with a dynamically replaceable kernel module [184].

Our goals are to be fast, support flexible policies, and provide high availability for applications. To achieve this, we need to address several challenges. First, we need to decouple marshalling from the application RPC library. Second, we need to design a new policy enforcement mechanism to process RPCs efficiently and securely, without incurring additional

marshalling overheads. Third, we need to provide a way for operators to specify/change policies and even change the underlying transport implementation without disrupting running applications.

We implement mRPC, the first RPC framework that follows the RPC as a managed service approach. Our results show that mRPC speeds up DeathStarBench [81] by up to 2.5×, in terms of mean latency, compared with combining state-of-art RPC libraries and sidecars, i.e., gRPC and Envoy, using the same transport mechanism. Larger performance gains are possible by fully exploiting network hardware capabilities from within the service. In addition, mRPC allows for live upgrades of its components while incurring negligible downtime for applications. Applications do not need to be re-compiled or rebooted to change policies or marshalling code. mRPC has three important limitations. First, data structures passed as RPC arguments must be allocated on a special shared-memory heap. Second, while we use a language-independent protocol for specifying RPC type signatures, our prototype implementation currently only works with applications written in Rust. Finally, our stub generator is not as fully featured as gRPC.

This work makes the following contributions:

- A novel RPC architecture that decouples marshalling/unmarshalling from RPC libraries to a centralized system service.
- An RPC mechanism that applies network policies and observability features with both security and low performance overhead, i.e., with minimal data movement and no redundant (un)marshalling. The mechanism supports live upgrade of RPC bindings, policies, transport, and marshalling without disrupting running applications.
- A prototype implementation of mRPC along with an evaluation on both synthetic workloads and real applications.

3.2 Background

In this section, we discuss the current RPC library architecture. We then discuss the emerging need for manageability and how manageability is implemented with existing RPC libraries.

3.2.1 Remote Procedure Call

To use RPC, a developer defines the relevant service interfaces and message types in a schema file (e.g., gRPC `.proto` file). A protocol compiler will translate the schema into program stubs that are directly linked with the client and server applications. To issue an RPC at runtime, the application simply calls the corresponding function provided by the stub; the stub is responsible for marshalling the request arguments and interacting with the transport layer (e.g., TCP/IP sockets or RDMA verbs). The transport layer delivers the packets to the remote server, where the stub unmarshals the arguments and dispatches the RPC request to a thread (eventually replying back to the client). We refer to this approach as *RPC-as-a-library*, since all RPC functionality is included in user-space libraries that are linked with each application. Even though the first RPC implementation [36] dates back to the 1980s, modern RPC frameworks (e.g., gRPC [93], eRPC [135], Thrift [247]) still follow this same approach.

A key design goal for RPC frameworks is efficiency. Google and Facebook have built their own efficient RPC frameworks, gRPC and Apache Thrift. Although primarily focused on portability and interoperability, gRPC includes many efficiency-related features, such as supporting binary payloads. Academic researchers have studied various ways to improve RPC efficiency, including optimizing the network stack [144, 293, 204], software hardware co-design [135, 137], and overload control [50].

As network link speeds continue to scale up [221], RPC overheads are likely to become even more salient in the future. This has led some researchers to advocate for direct application access to network hardware [212, 29, 135, 293], e.g., with RDMA or DPDK. Although low overhead, kernel bypass is largely incompatible with the need for flexible and enforceable layer 7 policy control, as we discuss next. In practice, multiple security weaknesses in RDMA hardware have led most cloud vendors to opt against providing direct access to RDMA by untrusted applications [264, 179, 231, 150, 151, 297].

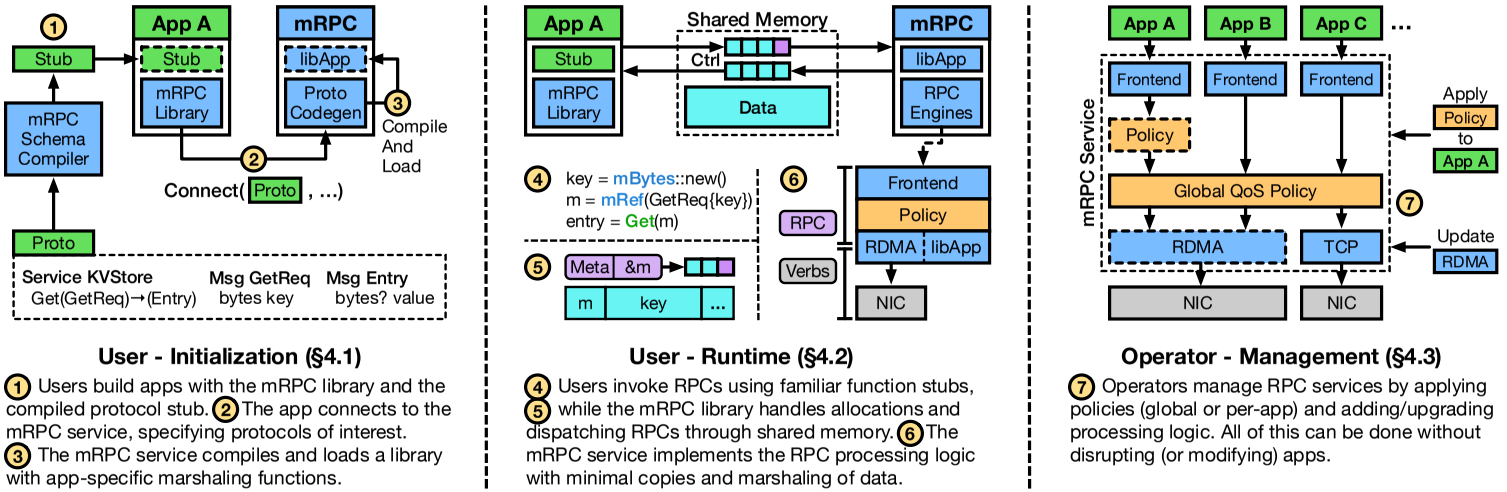


FIGURE 3.2: Overview of the mRPC workflow from the perspective of the users (and their applications) as well as infrastructure operators.

3.2.2 The Need for Manageability

As RPC-based distributed applications scale to large, complex deployment scenarios, there is an increasing need for improved manageability of RPC traffic. We classify management needs into three categories: **1) Observability:** Provide detailed telemetry, which enables developers to diagnose and optimize application performance. **2) Policy Enforcement:** Allow operators to apply custom policies to RPC applications and services (e.g., access control, rate limits, encryption). **3) Upgradability:** Support software upgrades (e.g., bug fixes and new features) while minimizing downtime to applications.

One natural question to ask is: *is it possible to add these properties without changing existing RPC libraries?* For observability and policy enforcement, the state-of-the-art solution is to use a sidecar (e.g., Envoy [68] or Linkerd [167]). A sidecar is a standalone process that intercepts every packet an application sends, reconstructing the application-level data (i.e., RPC), and applying policies or enabling observability. However, using a sidecar introduces substantial performance overhead, due to redundant RPC (un)marshalling. This RPC (un)marshalling, for example, in gRPC+Envoy, including HTTP framing and protobuf encoding, accounts for 62-73% overhead in the end-to-end latency [303]. In our evaluation (§3.6), using a sidecar increases the 99th percentile RPC latency by 180% and decreases the bandwidth by 44%. Figure 3.1a shows the (un)marshalling steps invoked as an RPC traverses from a client to a server and back. Using a sidecar triples the number of (un)marshalling steps (from 4 to 12). In addition, the sidecar approach is largely incompatible with the emerging trend of efficient application-level access to network hardware. Using sidecars means data buffers have to be copied between the application and sidecars, reducing the benefits of having zero-copy kernel-bypass access to the network.

Finally, using sidecars with application RPC libraries does not completely solve the upgradability issue. While policy can often be changed dynamically (depending on the feature set of the sidecar implementation), marshalling and transport code is harder to change. To fix a bug in the underlying RPC library, or merely to upgrade the code to take advantage

of new hardware features, we need to recompile the entire application (and sidecar) with the patched RPC library and reboot. gRPC has a monthly or two-month release cycle for bug fixes and new features [94]. Any scheduled downtime has to be communicated explicitly to the users of the application or has to be masked using replication; either approach can lead to complex application life-cycle management issues.

We do not see much hope in continuing to optimize this RPC library and sidecar approach for two reasons. First, a strong coupling exists between a traditional RPC library and each application. This makes upgrading the RPC library without stopping the application difficult, if not impossible. Second, there is only weak or no coupling between an RPC library and a sidecar. This prevents the RPC library and the sidecar from cross-layer optimization.

Instead, we argue for an alternative architecture in which RPC is provided *as a managed service*. By decoupling RPC logic, e.g., (un)marshalling, transport interface, from the application, the service can simultaneously provide high performance, policy flexibility, and zero-downtime upgrades.

3.3 Overview

Our system, mRPC, realizes the *RPC-as-a-managed-service* abstraction while maintaining similar end-to-end semantics as traditional RPC libraries (e.g., gRPC, Thrift). The goals for mRPC are to be fast, support flexible policy enforcement, and provide high availability for applications.

Figure 3.2 shows a high-level overview of the mRPC architecture and workflow, breaking it down into three major phases: initialization, runtime, and management. The mRPC service runs as a non-root, user-space process with access to the necessary network devices and a shared-memory region for each application. In each of the phases, we focus on the view of a single machine that is running both the RPC client application and the mRPC service. The RPC server may also run alongside an mRPC service. In this case, mRPC-specific marshalling can be used. However, we also support flexible marshalling to enable mRPC applications to interact with external peers using well-known formats (e.g., gRPC).

In our evaluation, we focus on cases where both the client and server employ mRPC.

The initialization phase extends from building the application to how the application binds to a specific RPC interface. (1) Similar to gRPC, users define a protocol schema. The mRPC schema compiler uses this to generate stub code to include in their application. We illustrate this using a key-value storage service with a single `Get` function. (2) When the application is deployed, it connects with the mRPC service running on the same machine and specifies the protocol(s) of interest, which are maintained by the generated stub. (3) The mRPC service also uses the protocol schema to generate, compile, and dynamically load a protocol-specific library containing the marshalling and unmarshalling code for that application's schemas[1]. This *dynamic binding* is a key enabler for mRPC to act as a long-running service, handling arbitrary applications (and their RPC schemas). [2]

At this point, we enter the runtime phase in which the application begins to invoke RPCs. Our approach uses *shared memory* between the application and mRPC, containing both control queues as well as a data buffer. (4) The application protocol stub produced by the mRPC protocol compiler can be called like a traditional RPC interface, with the exception that data structures passed as arguments or as return values must be allocated on a special heap in the shared data buffer. As an example, we show an excerpt of Rust-like pseudocode for invoking the `Get` function. (5) Internally, the stub and mRPC library manage RPC calls and replies in the control queues along with allocations and deallocations in the data buffer. (6) The mRPC service operates over the RPCs through modular *engines* that are composed to implement the per-application *datapaths* (i.e., sequence of RPC processing logic); each engine is responsible for one type of task (e.g., application interface, rate limiting, transport interface). Engines do not contain execution contexts, but are rather scheduled by *runtimes* in mRPC that correspond to kernel-level threads; during their execution, engines read from input queues, perform work, and enqueue outputs. External-facing engines (i.e., frontend, transport) use asynchronous control queues, while all other engines are executed

[1] Note that such libraries may be prefetched and/or cached to optimize the startup time.

[2] The dashed box of "Stub" and "libApp" means they are generated code.

synchronously by a runtime. Application control queues are contained in shared memory with the mRPC service.

This architecture, along with dynamic binding, enables mRPC to *operate over RPCs rather than packets*, avoiding the high overhead of traditional sidecar-based approaches. Additionally, the modular design of mRPC's processing logic enables mRPC to take advantage of fast network hardware (e.g., RDMA and smartNICs) in a manner that is transparent to the application. A key challenge, which we will address in §3.4.2, is how to securely enforce operator policies over RPCs in shared memory while minimizing data copies.

Finally, mRPC aims to improve the manageability of RPCs by infrastructure operators. Here, we zoom out to focus on the processing logic across all applications served by an mRPC service. ⑦ Operators may wish to apply a number of different policies to RPCs made by applications, whether on an individual basis (e.g., rate limiting, access control) or globally across applications (e.g., QoS). mRPC allows operators to add, remove, update, or reconfigure policies at runtime. This flexibility extends beyond policies to include those responsible for interacting with the network hardware. A key challenge, which we will address in §3.4.3, is in supporting the *live upgrade* of mRPC engines without interrupting running applications (and while managing engines sharing memory queues).

3.4 Design

In this section, we describe how mRPC provides dynamic binding, efficient policy and observability support, live upgrade, and security.

3.4.1 Dynamic RPC Binding

Applications have different RPC schemas, which ultimately decide how an RPC is marshalled. In the traditional RPC-as-a-library approach, a protocol compiler generates the marshalling code, which is linked into the application. In our design, the mRPC service is responsible for marshalling, which means that the application-specific marshalling code needs to be decoupled from an RPC library and run inside the mRPC service itself. Failing to ensure this separation would allow arbitrary code execution by a malicious user.

Applications directly submit the RPC schema (and not marshalling code) to the mRPC service. The mRPC service generates the corresponding marshalling code, then compiles and dynamically loads the library. Thus, we rely on our mRPC service code generator to produce the correct marshalling code for *any* user-provided RPC schema. For the initial handshake between an RPC client and an RPC server, the two mRPC services check that the provided RPC schemas match, and if not, the client's connection is rejected.

There are three remaining questions. First, **what are the responsibilities of the in-application user stub and mRPC library?** In mRPC, applications rely on user stubs to implement the abstraction as specified in their RPC schema. This means we still need to generate the glue code to maintain the traditional application programming interface. Our solution is to provide a separate protocol schema compiler, which is untrusted and run by application developers, to generate the user stub code that does not involve marshalling and transport. The application RPC stub (with the help of the mRPC library) creates a message buffer that contains the metadata of the RPC, with typed pointers to the RPC arguments, on the shared memory heap. The message is placed on a shared memory queue, which will be processed by the mRPC service. The receiving side works in a similar way.

Second, **does this approach increase RPC connect/bind time?** Implemented naively, this design will increase the RPC connect/bind time because the mRPC service has to compile the RPC schema and load the resulting marshalling library when an RPC client first connects to a corresponding server (or equivalently when an RPC server binds to the service). However, this latency is not fundamental to our design, and we can mitigate it in the following way. The mRPC service accepts RPC schemas before booting an application, as a form of prefetching. Given a schema, it compiles and caches the marshalling code. At the time of RPC connect/bind, the mRPC service simply performs a cache lookup based on the hash of the RPC schema. If it exists within the cache, the mRPC service will load the associated library; otherwise, the mRPC service will invoke the compiler to generate (and subsequently cache) the library. This reduces the connect/bind time from several seconds to several milliseconds.

Third, **when new applications arrive, do existing applications face downtime?** The multi-threaded mRPC service is a single process that serves many RPC applications; however, the marshalling engines for different RPC applications are not shared. They are in different memory addresses and can be (un)loaded independently. We will describe in §3.4.3 how to load/unload engines without disrupting running applications.

3.4.2 Efficient RPC Policy Enforcement and Observability

We have one key idea to allow efficient RPC policy enforcement and observability: senders should marshal once (as late as possible), while receivers should unmarshal once (as early as possible). On the sender side, we want to support policy enforcement and observability directly over RPCs from the application, and then marshal the RPC into packets. The receiver side is similar: packets should be unmarshalled into RPCs, applying policy and observability operations, and then delivered directly to the application. Compared to the traditional RPC-as-a-library approach with sidecars, this eliminates the redundant (un)marshalling steps (see Figure 3.1).

Data: DMA-capable shared memory heaps. Our design is centered around a dedicated shared memory heap between each application and the mRPC service. (Note that this heap is not shared across applications.) Applications directly create data structures, which may be used in RPC arguments, in a shared memory heap with the help of the mRPC library. Each application has a separate shared memory region, which provides isolation between (potentially mutually distrusting) applications. The mRPC library also includes a standard slab allocator for managing object allocation on this shared memory. If there is insufficient space within the shared memory, the slab allocator will request additional shared memory from the mRPC service and then map it into the application's address space. The mRPC service has access to the shared memory heap, allowing it to execute RPC processing logic over the application's RPCs, but also maintains a private memory heap for necessary copies.

Figure 3.3 shows an example workflow that includes access control for a key-value store service. Having the data structures directly in the shared memory allows an application

to provide pointers to data, rather than the data itself, when submitting RPCs to the mRPC service. We call the message sent from an application to the mRPC service an *RPC descriptor*. If there are multiple RPC arguments, the RPC descriptor points to an array of pointers (each pointing to a different argument on the heap).

Let us say we have an ACL policy that rejects an RPC if the key matches a certain string. The mRPC service first copies the argument (i.e., `key`), as well as all parental data structures (i.e., `GetReq`), onto its private heap. This is to prevent time-of-use-to-time-of-check (TOCTOU) attacks. Since applications have access to DMA-capable shared memory at all times, an application could modify the content in the memory while the mRPC service is enforcing policies. Copying arguments is a standard mitigation technique, similar to how OS kernels prevents TOCTOU attacks by copying system call arguments from user- to kernel-space. This copying only needs to happen if the policy behavior is based on the content of the RPC. We demonstrate in §3.6.2 that even with such copying, mRPC's overhead for an ACL policy is much lower than gRPC + Envoy. The RPC descriptor is modified so that the pointer to the copied argument now points to the private heap. On the receiver side, the TOCTOU attack is not relevant, but we need to take care not to place RPCs directly in shared memory. If there is a receive-side policy that depends on RPC argument values, the mRPC service first receives the RPC data into a private heap; it copies the RPC data into the shared heap after policy processing. This prevents the application from reading RPC data that should have been dropped or modified by the policies. Note that we can bypass this copy when processing does not depend on RPC argument values (e.g., rate limits). During ACL policy enforcement, the RPC is dropped if the `key` argument is contained in a blocklist. Note that if an RPC is dropped, any further processing logic is never executed (including marshalling operations).

Finally, at the end of the processing logic, the transport adapter engine executes. mRPC currently supports two types of transport: TCP and RDMA. For TCP, mRPC uses the standard, kernel-provided scatter-gather (`iovec`) socket interface. For RDMA, mRPC uses the scatter-gather `verb` interface, allowing the NIC to directly interact with buffers on the

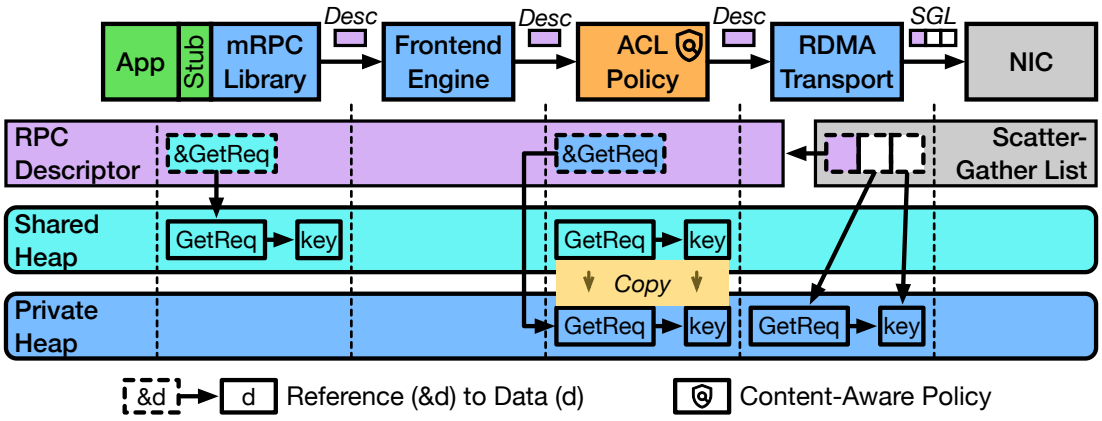


FIGURE 3.3: Overview of memory management in mRPC. Shows an example for the `Get` RPC that includes a content-aware ACL policy.

shared (or private) memory heaps containing the RPC metadata and arguments. For both TCP and RDMA, mRPC provides disjoint memory blocks to the transport layer directly, eliminating excessive data movements.[3]

Control: Shared-memory queues. To facilitate efficient communication between an application and the mRPC service, we use shared memory control queues. mRPC allocates two unidirectional queues for sending and receiving requests from an application to the mRPC service. The requests contain RPC descriptors, which reference arguments on the shared memory heap. The mRPC service always copies the RPC descriptors applications put in the sending queue to prevent TOCTOU attacks. mRPC provides two options to poll the queues: 1) busy polling, and 2) `eventfd`-based adaptive polling. In busy polling, both the application-side mRPC library and the mRPC service busy poll on their ends of the queues. In the `eventfd` approach, the mRPC library and the mRPC service sends event notifications after enqueuing to an empty queue. After receiving a notification, the queue is drained (performing the necessary work) before subsequently waiting on future events. The `eventfd` approach saves CPU cycles when queues are empty. Other alternative solutions may involve

[3] For RDMA, if the number of disjoint memory blocks exceeds the limit of NIC's capability to encapsulate all blocks in one RDMA work request, mRPC coalesces the data into a memory block before transmission. This is because sending a single work request (even with a copy) is faster than sending multiple smaller work requests on our hardware.

dynamically scaling up (or down) the number of threads used to busy poll by the mRPC service; however, we chose the `eventfd` approach for its simplicity. In our evaluation, we use busy polling for RDMA and eventfd-based adaptive polling for TCP.

Memory management. We provide a memory allocator in the mRPC library for applications to directly allocate RPC data structures to be sent on a shared memory heap. The allocator invokes the mRPC service to allocate shared memory regions on behalf of the application (similar to how a standard heap manager calls `mmap` or `sbrk` to allocate memory from an OS kernel). We need to use a specialized memory allocator for RPC messages (and their arguments), since RPCs are shared between three entities: the application, the mRPC service, and the NIC. A memory block is safe to be reclaimed only when it will no longer be accessed by any entity.

We adopt a notification-based mechanism for memory management. On the sender side, the outgoing messages are managed by the mRPC library within the application. On the receiver side, the incoming messages are managed by the mRPC service. When the application no longer accesses a memory block occupied by outgoing messages, the memory block will not be reclaimed until the library receives a notification from mRPC service that the corresponding messages are already sent successfully through the NIC (similar to how zero-copy sockets work in Linux). Incoming messages are put in buffers on a separate read-only shared heap. The receiving buffers can be reclaimed when the application finishes processing (e.g., when the RPC returns). To support reclamation of receive buffers, the mRPC library notifies the mRPC service when specific messages are no longer in use by the application. Notifications for multiple RPC messages are batched to improve performance. If the receiver application code wishes to preserve or modify the incoming data, it must make an explicit copy. Although this differs from traditional RPC semantics, in our implementation of Masstree and DeathStarBench we found no examples where the extra copy was necessary.

3.4.3 Live Upgrades

Although our modular engine design for the mRPC service is similar to Snap [179] and Click [149], we arrive at very different designs for upgrades. Click does not support live upgrades, while Snap executes the upgraded process to run alongside the old process. The old process serializes the engine states, transfers them to the new process, and the new process restarts them. This means that even changing a single line of code within a single Snap engine requires a complete restart for all Snap engines. This design philosophy is fundamentally not compatible with mRPC, as we need to deal with new applications arriving with different RPC schemas, and thus our upgrades are more frequent. In addition, we want to avoid fate sharing for applications: changes to an application's datapath should not impact the performance of other applications. Ultimately, Snap is a network stack that does not contain application-specific code, where as mRPC needs to be application-aware for marshalling RPCs.

We implement engines as plug-in modules that are dynamically loadable libraries. We design a live upgrade method that supports *upgrading, adding, or removing components of the datapath without disrupting other datapaths.*

Upgrading an engine. To upgrade one engine, mRPC first detaches the engine from its runtime (preventing it from being scheduled). Next, mRPC destroys and deallocates the old engine, but maintains the old engine's state in memory; note that the engine is detached from its queues and not running at this time. Afterwards, mRPC loads the new engine and configures its send and receive queues. The new engine starts with the old engine's state. If there is a change in the data structures of the engine's state, the upgraded engine is responsible for transforming the state as necessary (which the engine developer must implement). Note that this also applies to any shared state for cross-datapath engines. The last step is for mRPC to attach the new engine to the runtime.

Changing the datapath. When an operator changes the datapath to add or remove an engine, this process now involves the creation (or destruction) of queues and management

of in-flight RPCs. Changes that add an engine are straightforward, since it only involves detaching and reconfiguring the queues between engines. Changes that remove an engine are more complex, as some in-flight RPCs may be maintained in internal buffers; for example, a rate limiter policy engine maintains an internal queue to ensure that the output queue meets a configured rate. Engine developers are responsible for flushing such internal buffers to the output queues when the engines are removed.

Multi-host upgrades or datapath changes. Some engine upgrades or datapath changes that involve both the sender and the receiver hosts need to carefully manage in-flight RPCs across hosts. For example, if we want to upgrade how mRPC uses RDMA, both the sender and the receiver have to be upgraded. In this scenario, the operator has to develop an upgrade plan that may involve upgrading an existing engine to some intermediate, backward-compatible engine implementation. The plan also needs to contain the upgrade sequence, e.g., upgrading the receiver side before the sender side. Our evaluation demonstrates such a complex live upgrade, which optimizes the handling of many small RPC requests over RDMA (see §3.6.3).

3.4.4 Security Considerations

We envision two deployment models for mRPC: (1) a cloud tenant uses mRPC to manage its RPC workloads (similar to how sidecars are used today); (2) a cloud provider uses mRPC to manage RPC workloads on behalf of tenants. In both models, there are two different classes of principals: operators and applications. Operators are responsible for configuring the hardware/virtual infrastructure, deploying the mRPC service, and setting up policies that mRPC will enforce. Applications run on an operator's infrastructure, interacting with the mRPC service to invoke RPCs. Applications trust operators, along with all privileged software (e.g., OS) and hardware that the operators provide; both applications and operators trust our mRPC service and protocol compiler. In both deployment models, applications are not trusted and may be malicious (e.g., attempt to circumvent network policies).

In the first deployment model, mRPC service runs on top of a virtualized network that

is dedicated to the tenant. Running arbitrary policy and observability code inside the mRPC service cannot attack other tenants' traffic since inter-tenant isolation is provided by the cloud provider. In the second deployment model, our current prototype does not support running tenant-provided policy implementation inside mRPC service. How to safely integrate tenant-provided policy implementation and a cloud provider's own policy implementation is a future work.

From the application point of view, we want to ensure that **mRPC provides equivalent security guarantees as compared to today's RPC library and sidecar approach**, which we discuss in terms of: 1) dynamic binding and 2) policy enforcement. Our dynamic binding approach involves the generation, compilation, and runtime loading of a shared library for (un)marshalling application RPCs. Given that the compiled code is based on the application-provided RPC schema, this is a possible vector of attack. The mRPC schema compiler is trusted with a minimal interface: other than providing the RPC schema, applications have no control on the process of how the marshalling code is generated. We open source our implementation of the compiler so that it can be publicly reviewed.

As for all of our RPC processing logic, policies are enforced over RPCs by operating over their representations in shared memory control queues and data buffers. With a naive shared memory implementation, this introduces a vector of attack by exploiting a time-of-check to time-of-use (TOCTOU) attack; for instance, the application could modify the RPC message after policy enforcement but before the transport engine handles it. In mRPC, we address this by copying data into an mRPC-private heap prior to executing any policy that operates over the content of an RPC (as opposed to metadata such as the length). Similarly, received RPCs cannot be placed in shared memory until all policies have been enforced, since otherwise applications could see received RPCs before policies have a chance to drop (or modify) them. Shared memory regions are maintained by the mRPC service on a per-application basis to provide isolation.

Table 3.1: mRPC Engine Interface.

Operations
`doWork`(in:[Queue], out:[Queue]) *Operate over one or more RPCs available on input queues.*
`decompose`(out:[Queue]) → State *Decompose the engine to its compositional states.* *(Optionally output any buffered RPCs)*
`restore`(State) → Engine *Restore the engine from the previously decomposed state.*

3.5 Implementation

mRPC is implemented in 32K lines of Rust: 3K lines for the protocol compiler, 6K for the mRPC control plane, 12K for engine implementations, and 11K for the mRPC library. The mRPC control plane is part of the mRPC service that loads/unloads engines.

The mRPC control plane is not live-upgradable. The mRPC library is linked into applications and is thus also not live-upgradable. We do not envision the need to frequently upgrade these components because they only implement the high-level, stable APIs, such as shared memory queue communication and (un)loading engines.

Engine interface. Table 3.1 presents the essential API functions that all engines must implement. Each engine represents some asynchronous computation that operates over input and output queues via `doWork`, which is similar in nature to Rust's `Future`. mRPC uses a pool of runtime executors to drive the engines by calling `doWork`, where each runtime executor corresponds to a kernel thread. We currently implement a simple scheduling strategy inspired by Snap [179]: engines can be scheduled to a dedicated or shared runtime on start. In addition, runtimes with no active engines will be put to slept and release CPU cycles. The engines also implement APIs to support live upgrading: `decompose` and `restore`. In `decompose`, the engine implementation is responsible for destructing the engine and creating a representation of the final state of the engine in memory, returning a reference to mRPC. mRPC invokes `restore` on the upgraded instance of the engine, passing in a reference to the final state of the old engine. The developer is responsible for handling backward compati-

bility across engine versions, similar to how application databases may be upgraded across changes to their schemas.

Transport engines. We abstract reliable network communication of messages into transport engines, which share similar design philosophy with Snap [179] and TAS [144]. We currently implement two transport engines: RDMA and TCP. Our RDMA transport engine is implemented based on OFED libibverbs 5.4, while our TCP transport engine is built on Linux kernel's TCP socket.

3.5.0.0.1 mRPC Library. Modern RPC libraries allow the user to specify the RPC data types and service interface through a language-independent schema file (e.g., `protobuf` for gRPC, `thrift` for Apache Thrift). mRPC implements support for `protobuf` and adopts similar service definitions as gRPC, except for gRPC's streaming API. mRPC also integrates with Rust's async/await ecosystem for ease of asynchronous programming in application development.

To create an RPC service, the developer only needs to implement the functions declared in the RPC schema. The dependent RPC data types are automatically generated and linked with the application by the mRPC schema compiler. The mRPC library handles all the rest, including task dispatching, thread management, and error handling. To allow applications to directly allocate data in shared memory without changing the programming abstraction, we implement a set of shared memory data structures that expose the same rich API as Rust's standard library. This is done by replacing the memory allocation of data structures such as `Vec` and `String` with the shared memory heap allocator.

3.6 Evaluation

We evaluate mRPC using an on-premise testbed of servers with two 100 Gbps Mellanox Connect-X5 RoCE NICs and two Intel 10-core Xeon Gold 5215 CPUs (running at 2.5 GHz base frequency). The machines are connected via a 100 Gbps Mellanox SN2100 switch. Unless specified otherwise, we keep a single in-flight RPC to evaluate latency. To benchmark goodput and RPC rate, we let each client thread keep 128 concurrent RPCs on TCP and

32 concurrent RPCs on RDMA.

3.6.1 Microbenchmarks

We first evaluate mRPC's performance through a set of microbenchmarks over two machines, one for the client and the other for the server. The RPC request has a byte-array argument, and the response is also a byte array. We adjust the RPC size by changing the array length. RPC responses are an 8-byte array filled with random bytes. We compare mRPC with two state-of-the-art RPC implementations, eRPC and gRPC (v1.48.0). We deploy Envoy (v1.20) in HTTP mode to serve as a sidecar for gRPC. We use mRPC's TCP and RDMA backends to compare with gRPC and eRPC, respectively. There is no existing sidecar that supports RDMA. To evaluate the performance of using a sidecar to control eRPC traffic, we implement a single-thread sidecar proxy using the eRPC interface. We keep applications running for 15 seconds to measure the result.

Small RPC latency. We evaluate mRPC's latency by issuing 64-byte RPC requests over a single connection. Table 3.2 shows the latency for small RPC requests. Note that since the marshalling of small messages is fast on modern CPUs, the result in the table remains stable even when the message size scales up to 1 KB. We use `netperf` and `ib_read_lat` to measure raw round-trip latency.

mRPC achieves median latency of 32.8 µs for TCP and 7.6 µs for RDMA. Relative to netperf (TCP) or a raw RDMA read, mRPC adds 11.8 or 5.1 µs to the round-trip latency. This is the cost of the mRPC abstraction on top of the raw transport interface (e.g., socket, verbs).

We also evaluate latency in the presence of sidecar proxies. The sidecars do not enforce any policies, so we are only measuring the base overhead. Our results show that adding sidecars substantially increases the RPC latency. On gRPC, adding Envoy sidecars more than triples the median latency. The result is similar with eRPC. On mRPC, having a NullPolicy engine (which simply forwards RPCs) in the mRPC service has almost no effect on latency, increasing the median latency only by 300 ns.

Comparing the full solution (mRPC with policy versus gRPC/eRPC with proxy), mRPC speeds up the median latency by 6.1× (i.e., 33.4 µs against 203.4 µs) and the 99th percentile tail latency by 5.8×. On RDMA, mRPC speeds up eRPC by 1.3× and 1.4× in terms of median and tail latency (respectively). This is because the communication between the eRPC app and its proxy goes through the NIC, which triples the cost in the end-host driver (including the PCIe latency). In contrast, mRPC's architecture shortcuts this step with shared memory.

In addition, to separate the performance gain from system implementation difference, we evaluate the latency of mRPC with full gRPC-style marshalling (protobuf encoding and HTTP/2 framing) in the presence of NullPolicy engines as an ablation study. Under this setting, compared with gRPC + Envoy, mRPC speeds up the latency by 4.1× in terms of both median and tail latency. We also observe that the mRPC framework does not introduce significant overhead. Even with the cost of protobuf and HTTP/2 encoding, mRPC still achieves slightly lower latency compared with standalone gRPC. In mRPC, we can choose a customized marshalling format, because we know the other side is also an mRPC service. In other cases, e.g., when interfacing with external traffic or dealing with endianness differences, we can still apply full-gRPC style marshalling. When mRPC is configured to use full-gRPC style marshalling, we only need to pay (un)marshalling costs between mRPC services. For gRPC + Envoy, in addition to the (un)marshalling costs between Envoy proxies, the communication between applications and Envoy proxies also needs to pay this (un)marshalling cost. In the remaining evaluations, we will focus on mRPC's customized marshalling protocol.

Large RPC goodput. The client and server in our goodput test use a single application thread. The left side of Figure 3.4 shows the result. From this point on, when we discuss mRPC's performance, we focus on the performance of mRPC that has at least a NullPolicy engine in place to fairly compare with sidecar-based approaches.

mRPC speeds up gRPC + Envoy and eRPC + Proxy, by 3.1× and 9.3×, respectively, for 8KB RPC requests. mRPC is especially efficient for large RPCs[4], for which (un)marshalling

Table 3.2: **Microbenchmark [Small RPC latency]:** Round-trip RPC latencies for 64-byte requests and 8-byte responses.

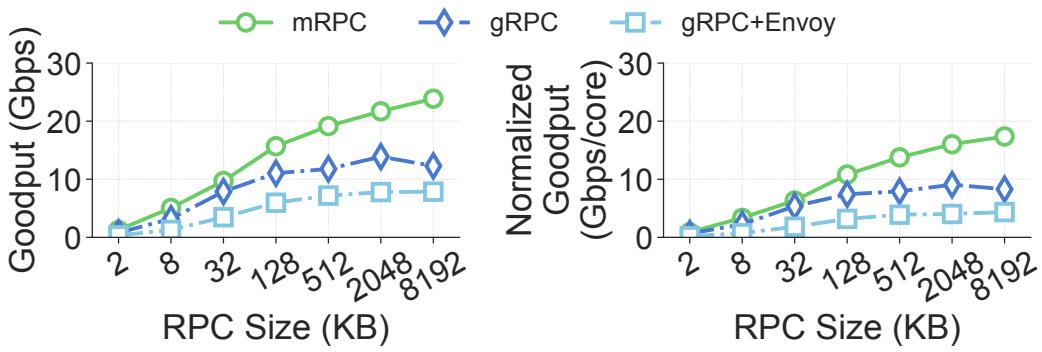
Transport	Solution	Median Latency (µs)	P99 Latency (µs)
TCP	Netperf	21.0	32.0
	gRPC	63.0	90.3
	mRPC	32.8	38.7
	gRPC+Envoy	203.4	251.1
	mRPC+NullPolicy	**33.4**	**43.3**
	mRPC+NullPolicy+HTTP+PB	49.8	61.9
RDMA	RDMA read	2.5	2.8
	eRPC	3.6	4.1
	mRPC	7.6	8.7
	eRPC+Proxy	11.3	15.6
	mRPC+NullPolicy	**7.9**	**9.1**

takes a higher fraction of CPU cycles in the end-to-end RPC datapath. Having a sidecar substantially hurts RPC goodput both for TCP and RDMA. In particular, for RDMA, intra-host roundtrip traffic through the RNIC might contend with inter-host traffic in the RNIC/PCIe bus, halving the available bandwidth for inter-host traffic. mRPC even outperforms gRPC (without Envoy). mRPC is fundamentally more efficient in terms of marshalling format: mRPC uses `iovec` and incurs no data movement.

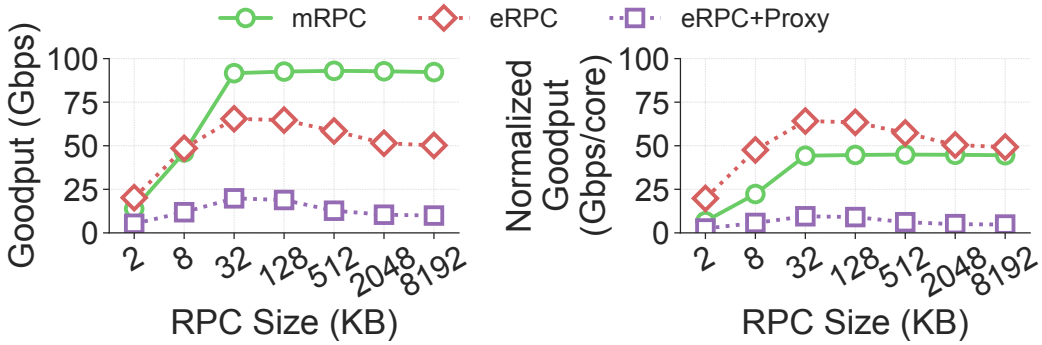
CPU overheads. To understand the mRPC CPU overheads, we measure the per-core goodput. The results are shown on the right side of Figure 3.4. mRPC speeds up gRPC + Envoy and eRPC + Proxy, by 3.8× and 9.3×, respectively. This means mRPC is much more CPU-efficient than gRPC + Envoy and eRPC + Proxy. eRPC (without a proxy) is quite efficient, but converges to mRPC's efficiency as RPC size increases.

RPC rate and scalability. We evaluate mRPC's small RPC rate and its multicore scalability. We fix the RPC request size to 32 bytes and scale the number of client threads. We use the same number of threads for the server as the client, and each client connects to one server thread. Figure 3.5 shows the RPC rates when scaling from 1 to 8 user threads. All the

[4] Standalone eRPC exhibits relatively lower goodput on RoCE than on Infiniband. According to the eRPC paper [135], eRPC should achieve 75 Gbps on Infiniband for 8MB RPCs.



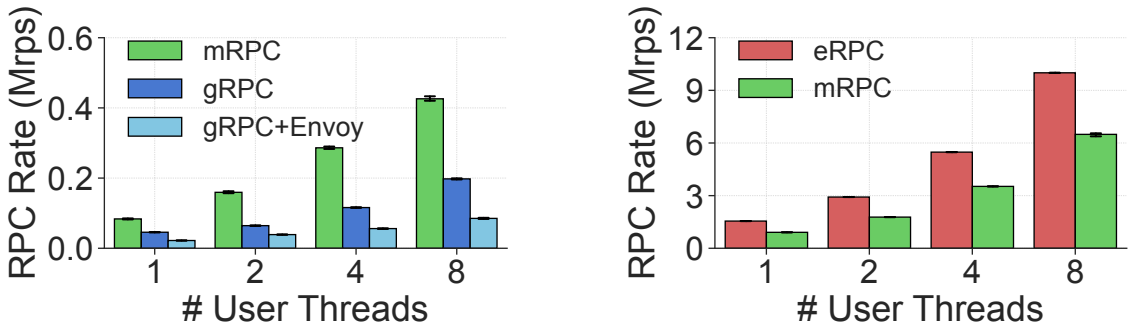
(a) TCP-based transport



(b) RDMA-based transport

FIGURE 3.4: **Microbenchmark [Large RPC goodput]:** Comparison of goodput for large RPCs. Note that different solutions demand different amounts of CPU cores, so we also normalized the goodput to their CPU utilization, as shown in the right figures. The error bars show the 95% confidence interval, but they are too small to be visible.

tested solutions scale well. mRPC's RPC rates scale by 5.1× and 7.2×, on TCP and RDMA, from a single thread to 8 threads. As a reference, gRPC scales by 4.3×, gRPC + Envoy scales by 3.9×, and eRPC scales by 6.5×. mRPC achieves 0.43 Mrps on TCP and 6.5 Mrps on RDMA with 8 threads. gRPC + Envoy only has 0.09 Mrps, so mRPC outperforms it by 5×. We do not evaluate eRPC + proxy, because our eRPC proxy is only single-threaded. When we run eRPC + proxy with a single thread, it achieves 0.51 Mrps. So even if eRPC + proxy scales linearly to 8 threads, mRPC still outperforms it.



(a) TCP-based transport

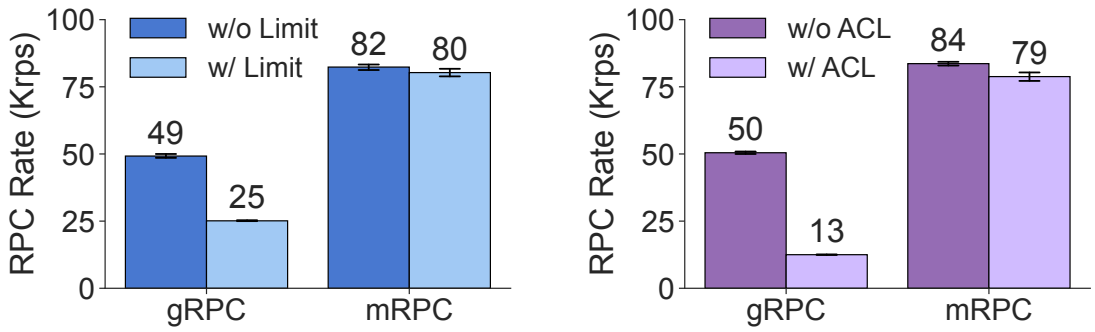
(b) RDMA-based transport

FIGURE 3.5: **Microbenchmark [RPC rate and scalability]:** Comparison of small RPC rate and CPU scalability. The bars show the RPC rate. The error bars show the 95% confidence interval.

3.6.2 Efficient Policy Enforcement

We use two network policies as examples to demonstrate mRPC's efficient support for RPC policies: (1) RPC rate limiting and (2) access control based on RPC arguments. RPC rate limiting allows an operator to specify how many RPCs a client can send per second. We implement rate limiting as an engine using the token bucket algorithm [257]. Our access control policy inspects RPC arguments and drops RPCs based on a set of rules specified by network operators. These two network policies differ greatly from traditional rate limiting and access control, which only limit network bandwidth and can only operate on packet headers.

We compare rate limit enforcement using an mRPC policy versus using Envoy's rate limiter on gRPC workloads. To evaluate the performance overheads, we set the limit to infinity so that the actual RPC rate is never above the limit (allowing us to observe the overheads). Figure 3.6a shows the RPC rate with and without the rate limits. gRPC's RPC rate drops immediately from 49K to 25K. This is because having a sidecar proxy (Envoy) introduces substantial performance overheads. For mRPC, the RPC rate stays the same at 82K. This is because having a policy introduces minimal overheads. The extra policy only adds tens to hundreds of extra CPU instructions on the RPC datapath.



(a) Rate Limiting (b) Access Control

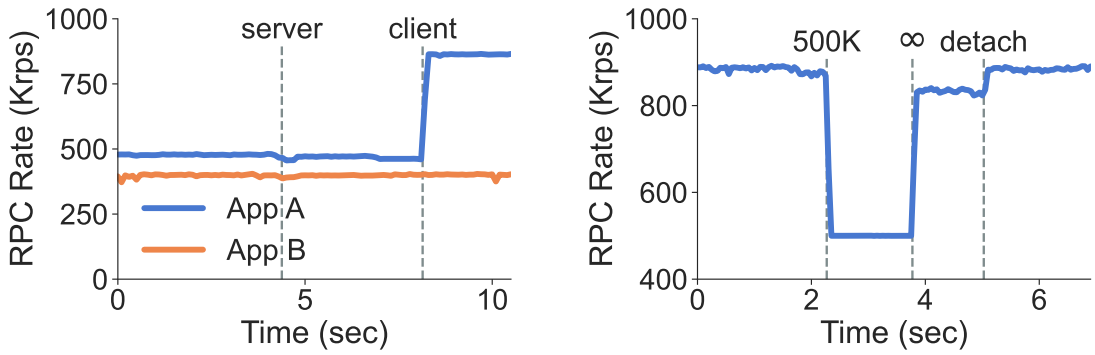
FIGURE 3.6: **Efficient Support for Network Policies**. The RPC rates with and without policy are compared. The bars of w/o Limit and w/o ACL for gRPC show its throughput when the sidecar is bypassed. The error bars show the 95% confidence interface.

We evaluate access control on a hotel reservation application in DeathStarBench [81]. The service handles hotel reservation RPC requests, which include the customer's name, the check-in date, and other arguments. The service then returns a list of recommended hotel names. We set the access control policy to filter RPCs based on the `customerName` argument in the request. We use a synthetic workload containing 99% valid and 1% invalid requests. We again compare our mRPC policy against using Envoy to filter gRPC requests. We implement the Envoy policy using WebAssembly. gRPC's rate drops from 50K to 13K. This is because of the same sidecar overheads and now Envoy has to further parse the packets to fetch the RPC arguments. On mRPC, the performance drop is much smaller, from 84K to 79K. Note that, on mRPC, the performance overhead of introducing access control is larger than rate limiting. For access control, the mRPC service has to copy the relevant field (i.e., `customerName`) to the private heap to prevent TOCTOU attacks on the sender side and has to copy the RPC from a private heap to the shared heap on the receiver side.

3.6.3 Live Upgrade

We demonstrate mRPC's ability to live upgrade using two scenarios.

Scenario 1. During our development of mRPC, we realized that using the RDMA NIC's scatter-gather list to send multiple arguments in a single RPC can significantly boost



(a) Transport adapter

(b) Rate limit policy

FIGURE 3.7: **Live upgrade**. In (a), the annotations indicate when the client of App A and server of A and B are upgraded. In (b), the annotations denote the specified rate and when the policy is removed.

mRPC's performance. In this approach, even when an RPC contains arguments that are scattered in virtual memory, we can send the RPC using a single RDMA operation (`ibv_post_send`). We use these two versions of our RDMA transport engine to demonstrate that mRPC enables such an upgrade without affecting running applications. Note that all other evaluations already include this RDMA feature. This upgrade involves both the client side's mRPC service and the server side's mRPC service, because it involves how RDMA is used between machines (i.e., transport adapter engine). gRPC and eRPC cannot support this type of live upgrade.

We run two applications (App A and App B). Both applications are sending 32-byte RPCs, and the responses are 8 bytes. A and B share the mRPC service on the server side. A's and B's RPC clients are on different machines. We keep 8 concurrent RPCs for B, forcing it to send at a slower rate, while using 32 for A. We first upgrade the server side to accept arguments as a scatter-gather list, and we then upgrade the client side of A. Figure 3.7a shows the RPC rate of A and B. When the server side upgrades, we observe a negligible effect on A's and B's rate. Neither A nor B needs recompilation or rebooting. When A's client side's mRPC service is upgraded, A's performance increases from 480K to 860K. B's performance is not affected at all because B's client side's mRPC service is not upgraded.

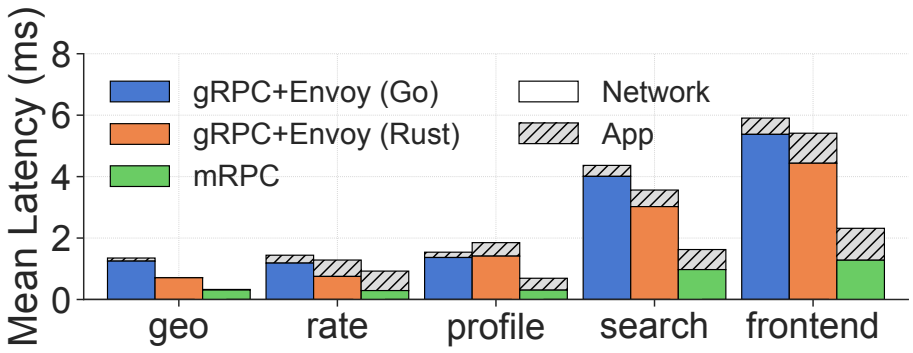


FIGURE 3.8: **DeathStarBench**: Mean latency of in-application processing and network processing of microservices. The latency of a microservice includes RPC calls to other microservices. The frontend latency represents complete end-to-end latency.

Scenario 2. Enforcing network policies has performance overheads, even when they do not have any effect. For example, enforcing a rate limit of an extremely large throttle rate still introduces performance overheads just for tracking the current rate using token buckets. mRPC allows policies to be removed at runtime, without disrupting running applications.

We use the same rate limiting setup from §3.6.2 but on top of RDMA transport. Figure 3.7b shows the RPC rate. We start from not having the rate limit engine. We then load the rate limit engine and set the throttled rate to 500K. The RPC rate immediately becomes 500K. We then set the throttled rate to be infinite, and the rate becomes 840K. After we detach the rate limit engine, the rate becomes 890K.

Takeaways. There are two overall takeaways from these experiments. First, mRPC allows upgrades to the mRPC service without disrupting running applications. Second, live upgrades allow for more flexible management of RPC services, which can be used to enable immediate performance improvements (without redeploying applications) or dynamic configuration of policies.

3.6.4 Real Applications

We evaluate how the performance benefits of mRPC transform into end-to-end application-level performance metrics.

DeathStarBench. We use the hotel reservation service from the DeathStarBench [81] microservice benchmark suite. The reference benchmark is implemented in Go with gRPC and Consul [55] (for service discovery). Our mRPC prototype currently only supports Rust applications, and we thus port the application code to Rust for comparison. We use the same open-source services such as memcached [182] and MongoDB [189].

We distribute the HTTP frontend and the microservices on four servers in our testbed. The monolithic services (memcached, MongoDB) are co-located with the microservices that depend on them. We use a single thread for each of the microservices and the frontend. Further, we deploy an Envoy proxy as a sidecar on each of the servers (with no active policy). The provided workload generator [81] is used to submit HTTP requests to the frontend. For a fair comparison, we also implemented a Rust version of the benchmark with Tonic [260], which is the de facto implementation of gRPC in Rust. We deploy the mRPC and Tonic implementations on bare metal, while the reference Go suite runs in Docker containers with a host network (which introduces negligible performance overheads compared to using bare metal [309]). All three solutions are based on TCP. We issue 20 requests per second for 250 seconds and record the latency of each request, breaking it down into the in-application processing time and network processing time for each microservice involved. In our evaluation, the dynamic bindings of the user applications are already cached in mRPC service, so the time to generate the bindings is not included in the result.

Figure 3.8 shows the latency breakdown. First, we validate that our own implementation of DeathStarBench on Rust is a faithful re-implementation. We can see that the original Go implementation and our Rust implementation have similar latency. Moreover, the amount of latency spent in gRPC is similar. Second, mRPC with a null policy outperforms by 2.5× gRPC with a sidecar proxy in average end-to-end latency.

Masstree analytics. We also evaluate the performance of Masstree [177], an in-memory key-value store, over both mRPC and eRPC [135] using RDMA. We follow the exact same workload setup used in eRPC, which contains 99% I/O-bounded point GET request and 1% CPU-bounded range SCAN request. We run the Masstree server on one machine and

Table 3.3: **Masstree analytics:** Latency and the achieved throughput for GET operations. MOPS is Million Operations Per Second.

	Median Latency	P99 Latency	Throughput
eRPC	16.8 µs	21.7 µs	8.7 MOPS
mRPC	22.5 µs	33.1 µs	7.0 MOPS

run the client on another machine. Both the server and the client use 10 threads, with each client thread using 16 concurrent requests. The test runs for 60 seconds. The result in Table 3.3 shows that eRPC outperforms mRPC, which makes sense since eRPC is a well-designed library implementation that is focused on high performance. mRPC enables many other manageability features in exchange for a slight reduction in performance. In this case, using mRPC instead of eRPC means that median latency increases by 34% and throughput reduces by 20%.

3.7 Related Work

3.7.0.0.1 Fast RPC implementations. Optimizing RPC has a long history. Birrell and Nelson's early RPC design [36] includes generating bindings via a compiler, interfacing with transport protocols, and various optimizations (e.g., implicit ACK). Bershad et al. showed how to use shared-memory queues to efficiently pass RPC messages between processes on the same machine [34]. mRPC's shared-memory region leverages this idea but extends it to allow for marshalling code to be applied after policy enforcement. A similar use of shared-memory queues can be found with recent Linux support for asynchronous system calls [17] combined with scatter-gather I/O [172]; unlike traditional system calls, however, mRPC protocol descriptions can be defined at runtime.

Another line of work uses RDMA to speed network RPCs [252, 253, 47, 188, 137, 135]. These studies assume direct application access to network hardware and are thus susceptible to RDMA's security weaknesses [231]. mRPC leverages ideas from RDMA RPC research but in a way that is compatible with policy enforcement and observability, by doing so as a service. Another line of work reduces the cost of marshalling, by using alternative

formats [223, 259, 74, 41, 35, 132, 195, 12] or designing hardware accelerators [139, 219, 278, 123]. This work is largely orthogonal to our goal of removing unnecessary marshalling steps but could be applied to further improve mRPC performance.

3.7.0.0.2 Fast network stacks. Building efficient host network stacks is a popular research topic. MegaPipe [101], mTCP [128], Arrakis [212], IX [29], eRPC [135], and Demikernel [293] advocate building the network stack as a user-level library, bypassing the kernel for performance. In these systems, an application directly accesses the network interface, but they assume policy can be enforced by the network hardware and are thus vulnerable if the hardware has security weaknesses. mRPC can interpose policy on any RPC. Like mRPC, Snap [179] and TAS [144] implement the network stack as a service, but they stop at layer 4 (TCP and UDP) rather than layer 7 (RPC). Application RPC stubs must marshal data into shared memory queues to use Snap or TAS. Flexible policy engines are a key feature of Snap, but because Snap operates at layer 4, it can only apply layer 7 policies by unmarshalling and re-marshalling RPC data. A fast network stack like mRPC can also be implemented directly in the kernel. LITE [264] implements RDMA operations as system calls inside the kernel to improve manageability, and Shenango [204] interposes a specialized kernel packet scheduler for network messages.

3.7.0.0.3 Fast network proxies. There is a long line of work on improving the performance of network proxies [149, 213, 248, 206, 117, 207, 122, 141, 178, 159, 146, 183, 215, 294]. Much of this work considers the general case of a standalone proxy. Our work differs in two ways. First, our proposed technique is only for RPC traffic rather than generalized TCP traffic. Second, we co-design the application library stub and proxy, and thus, both must be co-located on the same machine for our shared memory queues to function. In today's sidecar proxies (our baseline), this assumption holds, but it does not hold for generalized network proxies.

3.7.0.0.4 Live upgrades of system software. Being able to update system software without disrupting or restarting applications is key to achieving end-to-end high availability.

Snap [179] provides live upgrade of the network stack running as a proxy; Bento [184] provides similar functionality for kernel-resident file systems. Relative to these systems, mRPC upgrades are more fine-grained. For example, Snap targets a maximum outage during upgrades of 200 milliseconds, by spawning another instance of itself and moving all connections to the new process. By contrast, our goal is near instantaneous changes and upgrades to RPC protocol definitions, policy engines, and marshalling code. We accomplish this by keeping the control plane intact and performing updates by loading and unloading dynamic libraries. eBPF is a Linux kernel extensibility mechanism that supports dynamic updates [67]; unlike eBPF, mRPC can dynamically change the execution graph of policy engines as well as the individual engines themselves.

3.8 Summary

Remote procedure call has become the de facto abstraction for building distributed applications in datacenters. The increasing demand for manageability makes today's RPC libraries inadequate. Inserting a sidecar proxy into the network datapath allows for manageability but slows down RPC substantially due to redundant marshalling and unmarshalling. We present mRPC, a novel architecture to implement RPC as a managed service to achieve both high performance and manageability. mRPC eliminates the redundant marshalling overhead by applying policy to RPC data before marshalling and only copying data when necessary for security. This new architecture enables live upgrade of RPC processing logic and new RPC scheduling and transport methods to improve performance. We have performed extensive evaluations through a set of micro-benchmarks and two real applications to demonstrate that mRPC enables a unique combination of high performance, policy flexibility, security, and application-level availability. Our source code is available at `https://github.com/phoenix-dataplane/phoenix`.

4. NUSE: Towards a FUSE Counterpart for Networking

In the previous chapters, we demonstrated concrete benefits of co-designing networking with applications through two targeted systems, NetHint and mRPC. We now broaden our scope to consider a more general approach to extensibility in the operating system's network stack. NetHint and mRPC each tackled specific limitations: the former provided applications with network insight, and the latter moved application-level communication logic into the OS for efficiency; chapter 4 takes a step further by asking how the operating system itself can be rearchitected to better support such application-networking integration. In this chapter, we introduce NUSE, which can be viewed as a networking analogue to FUSE (Filesystem in Userspace). The goal of NUSE is to enable flexible experimentation and deployment of network functionality in user space while still leveraging the kernel for performance-critical operations. This transition marks a shift from solving today's problems to anticipating future needs: we move from improving existing interfaces to proposing a new architecture for host networking.

4.1 Introduction

Host networking has been a focal point of continuous innovation in both academia and industry, fueled by the proliferation of performance-critical workloads such as large language models (LLMs) [203, 263, 48], data analytics [60, 288, 42, 262, 173, 90], and other cloud-scale services [111, 274]. Over the past decade, researchers and practitioners have explored numerous strategies to optimize host networking, including transport optimization [7, 276, 304, 185, 190, 153], flow scheduling and load balancing [18, 53, 291], enhancements in packet processing and network virtualization [213, 178, 309, 108], drawing on a wide range of techniques spanning kernel-level enhancements, kernel-bypass frameworks, and programmable hardware.

These efforts underscore the importance of reducing software overheads and improving performance isolation, yet they also highlight the *fragmentation* of approaches in this space. While some solutions rely on extensive kernel modifications [205, 18, 276], requiring spe-

cialized developer expertise and careful maintenance, others build entirely new user-level stacks [128, 144, 66, 1] or introduce hardware offloading mechanisms [147, 72, 24]. This diversity in solutions showcases the creativity of the research community but also results in a scattered ecosystem. Consequently, there is no single, standardized methodology for implementing and comparing advanced host networking ideas, making it difficult for researchers to evaluate their innovations in a common context.

Building and deploying host networking solutions is challenging. In many high-speed environments, overhead in the kernel TCP stack [246, 39] can overshadow the performance gains promised by new designs, limiting their practical impact. Integrating a new feature into the Linux kernel requires working through complex APIs, patch reviews, and version-specific constraints (given the kernel does not guarantee a stable API), which slows down research and prototyping. Some projects by pass the kernel entirely, but user-level or hardware-specific approaches often lack broad adoption and can struggle to provide centralized subsystems, such as firewall or traffic control, which are essential for a networked system. A lack of standardized benchmarking adds another layer of difficulty. Researchers use different frameworks, hardware configurations, and workload scenarios, making their results difficult to directly compare. Consequently, a promising idea might only be tested under narrow conditions, creating uncertainty about its benefits in other settings. Code reusability is also hampered by closed-source licensing or specialized dependencies, discouraging collaboration. Without a common platform for easy implementation and evaluation, many host networking optimizations remain limited to specialized deployments or academic prototypes, and never reach their full potential in broader practice.

To address these challenges, we propose a unified experimental platform, NUSE, that aims to streamline host networking research by lowering the barriers to prototyping, evaluation, and deployment. We draw inspiration from Filesystem in userspace (FUSE) [62], which has been widely adopted in production and, in some cases, has even outperformed traditional in-kernel file systems in terms of flexibility and usability. The success of FUSE demonstrates that a well-designed user-space framework can gain widespread adoption while

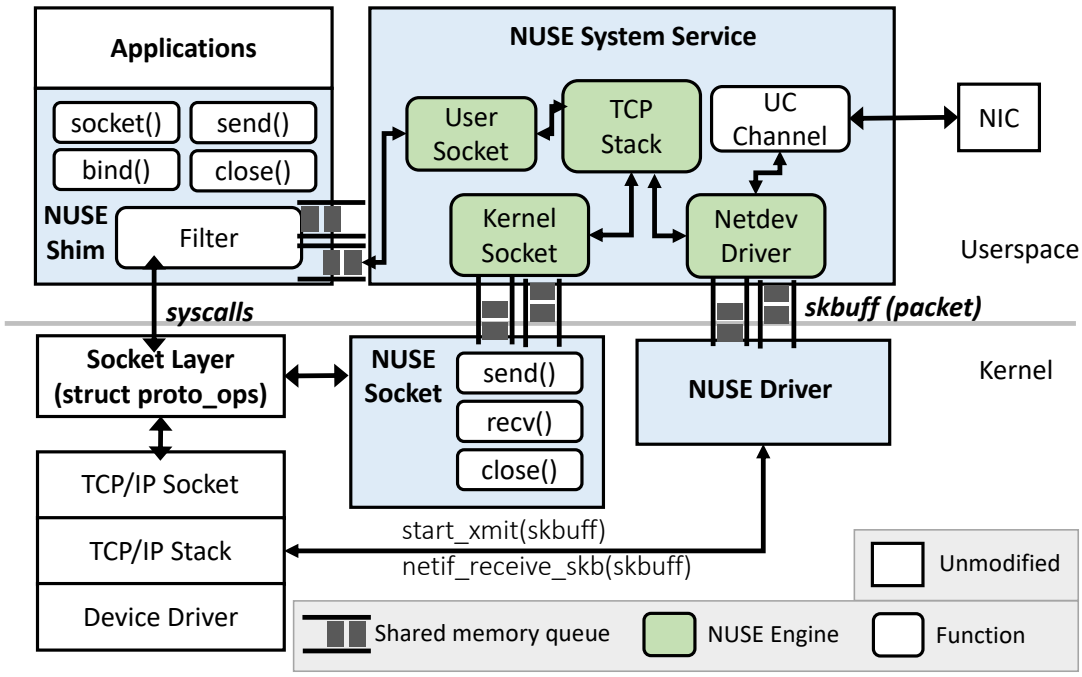


FIGURE 4.1: The NUSE hybrid architecture. The NUSE Shim intercepts network-related system calls from an application and directs them to the user-space NUSE System Service. The NUSE System Service contains multiple engines (for user sockets, kernel compatibility, a TCP stack, and a `netdev` driver) to process network operations in user space. The NUSE Driver in the kernel forwards packets between the kernel's networking stack and the user-space stack, acting as a virtual network interface. The UC Channel leverages RDMA (via `ibverbs`) to offload data transmission from user space directly to the NIC, achieving high throughput with minimal kernel intervention.

maintaining high performance. Similarly, an effective user-space networking platform should provide the same level of extensibility and practicality as FUSE. By adopting a hybrid-kernel architecture—drawing inspiration from Snap [179, 144, 45, 281], developers can flexibly relocate host networking logic (for instance, device-driver or transport-layer components) into userspace, while continuing to leverage the kernel stack for any unmodified functionality. This design balances ease of experimentation with the robustness of the kernel's networking stack, reducing development complexity while addressing fragmentation. In addition, NUSE includes standardized benchmarking tools, enabling consistent evaluation of different designs under common and representative workloads. We provide ready-to-run scripts and application profiles (e.g., Redis, Memcached, and collective communication applications), so

research can easily measure the impact of their host networking improvements.

Although NUSE draws inspiration from FUSE, there are several key distinctions between the two. In FUSE, for simplicity, all file system-related system calls enter the kernel, with some calls being forwarded to userspace for handling. However, because a file system provides a block device-like API, certain requests—such as `read` or `statfs`—can be cached and served directly from the kernel when the underlying files remain unchanged. As a result, under read-most workloads, FUSE can achieve performance comparable to in-kernel file systems due to kernel caching. In contrast, networking operates under a fundamentally different paradigm. The networking subsystem provides a streaming-like API, where caching opportunities are minimal or nonexistent. Simply replicating FUSE's architecture would lead to significant performance bottlenecks. To address this challenge, NUSE adopts a hybrid approach based on `LD_PRELOAD` and in-shim filtering mechanisms. Specifically, NUSE provides a virtual network device (`netdev`), which networked applications explicitly bind to when allocating socket file descriptors. All socket requests that match the NUSE device and protocol are redirected via a ring buffer to the NUSE system service for processing, while non-relevant sockets are handled directly by the kernel. For statically linked applications (such as most Go programs), where dynamic function interposition via `LD_PRELOAD` is ineffective, system calls first pass through the kernel before being redirected to NUSE —similar to how FUSE handles requests in cases where userspace interception is unavailable. While this introduces an additional performance overhead, it ensures broad compatibility with existing applications.

This work explores three key questions. First, is such a system feasible within a hybrid-kernel architecture, and what are the major challenges in implementing it? We outline the design of NUSE in §4.3 and discuss key implementation blockers in §4.4. Second, what are the trade-offs and challenges in realizing this approach? We analyze the implications of shared memory queues, data movement, and application transparency as fundamental constraints in §4.2. Finally, where does this approach apply, and what are its limitations? We explore NUSE's applicability, potential constraints, and future directions in §4.5.

4.2 Background and Motivation

Despite significant advancements, host networking research remains fragmented, making it difficult for researchers to prototype, evaluate, and deploy new designs in a unified environment. Existing solutions vary widely, from kernel modifications to user-space stacks and hardware offloading, yet these approaches are often incompatible, leading to isolated development efforts. The lack of a standardized evaluation framework further complicates progress, as researchers rely on diverse testbeds, workloads, and performance metrics, making direct comparisons between techniques unreliable. Moreover, integrating innovations into real-world systems remains challenging—kernel modifications require navigating complex APIs and version constraints, while user-space alternatives often lack system-wide integration, limiting their practical adoption.

4.2.1 Limitations of Existing Approaches

Efforts to improve host networking largely fall into two categories: kernel-bypass approaches and in-kernel extensibility.

kernel-bypass techniques, including user-space library network stacks and frameworks such as DPDK and RDMA, eliminate kernel overhead by processing packets in user space, they introduce deployment challenges. These solutions require applications to be rewritten or linked against custom libraries, reducing compatibility with existing software. Additionally, they often lack integration with system-wide services like firewalling and traffic control, forcing developers to reimplement essential functionality. While frameworks such as DPDK deliver high throughput, they depend on specialized hardware and require non-trivial application modifications, making them impractical for general-purpose networking research.

In-kernel extensibility mechanisms, such as eBPF, offer a way to modify networking behavior without kernel patches, allowing safe and dynamic updates to packet processing logic. However, eBPF programs are constrained by strict verification rules, limiting their expressiveness and preventing full transport-layer modifications. Although eBPF provides an efficient means for extending kernel behavior, it remains unsuitable for designing entirely

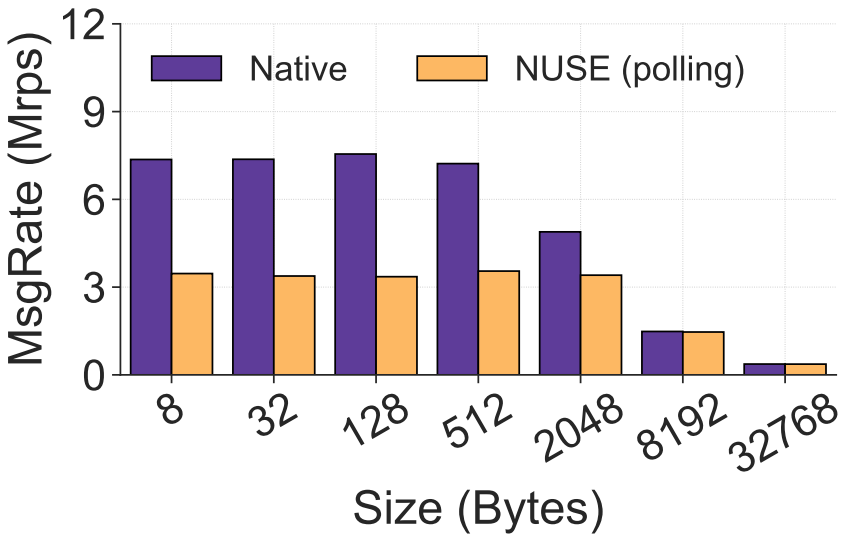


FIGURE 4.2: Message rate comparison between Native RDMA WRITE and NUSE (polling). While NUSE achieves near-native performance for larger messages, smaller messages show a significant gap due to system overhead.

new networking architectures.

4.2.2 Performance Implications of Shared Memory Queues

To understand the performance implications of NUSE's architecture, we evaluate a micro-benchmark scenario in which an application continuously sends fixed-size messages. To isolate system overhead, rather than using the socket API, we implement a prototype that forwards RDAM Verbs calls to the NUSE system service, an even more performance-sensitive use case. The experiments are conducted on two servers, each with one 100 Gbps Mellanox Connect-X5 RoCE NIC and two Intel 10-core Xeon Gold 5215 CPUs. We use perftest, a standard RDMA benchmarking tool, with RDMA WRITE as the baseline comparison. Each thread maintains a single Queue Pair (QP)/connection to ensure fair comparison across configurations.

Figure Figure 4.2 presents the message rate as a function of message size. When the message size is small, NUSE achieves 3.5 million RPS (requests per second), while native RDMA WRITE reaches 7.3 million RPS, nearly double the throughput. As message size increases towards 8 KB, the message rates converge due to bandwidth becoming the primary

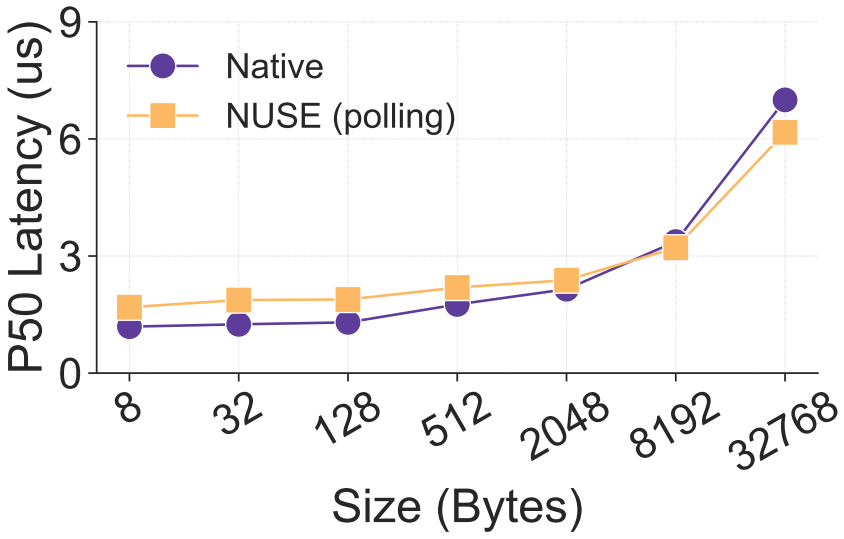


FIGURE 4.3: Latency comparison between Native RDMA WRITE and NUSE (polling). NUSE incurs a small but measurable overhead for small messages, which diminishes as message size increases.

bottleneck rather than system overhead.

Another key observation is CPU efficiency: while NUSE achieves comparable bandwidth for large messages, it consumes twice the CPU resources compared to native RDMA. This means that when measuring per-core message rate, NUSE achieves only one-fourth of the efficiency of native RDMA.

Figure Figure 4.3 illustrates the median latency across different message sizes. For small messages, NUSE exhibits a median latency of 1.9 µs, compared to 1.3 µs for native RDMA WRITE, indicating an additional overhead of approximately 600 ns. This overhead, which stems from shared memory polling and software stack processing, remains relatively small even under extreme conditions. As message size grows beyond 2 KB, this additional latency is effectively hidden by transmission time, making NUSE's latency indistinguishable from the native baseline.

Implications for Socket-Based Applications. When extending these results to socket-based communication, we note that traditional BSD sockets inherently introduce an overhead of 1µs per call, regardless of message size. Given this baseline, NUSE's additional overhead

for sockets is expected to be negligible, as its primary impact is within sub-microsecond latency ranges.

However, the use of busy polling in shared memory queues results in twice the CPU consumption. This suggests two potential optimization paths: (1) Reducing CPU polling overhead through advanced mechanisms, such as event-driven wake-ups or user interrupts (§4.4). (2) Employing event notification mechanisms for latency-insensitive workloads, selectively using polling for performance-critical paths.

4.2.3 Motivation for a Hybrid-Kernel Approach

Given these limitations and the performance implications of shared memory queues, we propose a hybrid-kernel architecture that balances user-space flexibility with kernel compatibility while leveraging RDMA NICs for performance scalability. As RDMA has become increasingly common, especially in deep learning and large-language model workloads, it provides a natural foundation for high-performance host networking. However, existing RDMA and kernel-bypass frameworks often require extensive rewrites and specialized expertise, limiting their adoption.

Our approach avoids this complexity by enabling transparent RDMA acceleration without requiring applications to be restructured or a full user-space stack to be reimplemented. Unlike eBPF, which is constrained in modifying transport-layer behavior, our design supports deeper stack customization while retaining integration with system-wide networking features. This enables researchers to experiment with new transport mechanisms efficiently while ensuring compatibility with existing workloads.

4.3 NUSE Design

NUSE introduces a hybrid-kernel architecture that intercepts standard socket calls from applications and routes them to a user-space networking stack when appropriate. This design provides a user-space execution environment for transport-layer protocols, reducing kernel overhead while still allowing flexibility and extensibility. As shown in Figure 4.1 The core components of NUSE include the NUSE Shim, NUSE System Service, NUSE Driver,

and UC Channel – these work together to optimize data movement and CPU utilization in networking operations.

4.3.1 NUSE Shim: Dispatching System Calls

To enable transparent redirection of networking calls, NUSE uses `LD_PRELOAD` to interpose on socket-related system calls such as `socket()`, `send()`, `recv()`, and `bind()`. The NUSE Shim library intercepts these calls and filters them based on the application's requirements, ensuring that only relevant traffic is diverted to the user-space stack. Non-relevant traffic (for example, DNS or DHCP requests) is passed through to the standard kernel networking stack as usual. In cases where dynamic interception is ineffective (e.g. in statically linked Go binaries), the system calls follow a fallback path into the kernel and are then forwarded back to NUSE for processing, ensuring compatibility even when direct interception isn't possible.

4.3.2 NUSE System Service: Userspace Network Processing

The NUSE System Service is a user-space daemon responsible for handling all intercepted networking operations. It consists of multiple *NUSE Engines*, each managing a different aspect of the networking stack to maximize parallelism and modularity.

User Socket Engine pools a shared memory queue for intercepted socket operations coming from applications and dispatches them for processing.

Kernel Socket Engine handles any operations that had to fall back to the kernel, providing a compatibility layer so that those operations can still be managed with minimal disruption.

TCP Stack Engine implements transport-layer protocol logic (e.g. a TCP/IP stack in user space). This engine is modular, allowing researchers to plug in custom transport protocols such as Homa or lightweight TCP implementations (for example, a Rust `smoltcp` library) without reimplementing common networking functionality from scratch.

Netdev Driver Engine bridges the userspace stack with the kernel's network interface layer. It batches outgoing packets (as `sk_buffs`) and sends them through shared memory to the kernel, reducing the overhead of frequent syscalls for packet transmission.

By splitting responsibilities across these engines, the NUSE System Service can efficiently process network I/O in user space while cooperating with the kernel when needed.

4.3.3 NUSE Driver: Kernel-Space Virtual Device

The NUSE Driver is a kernel-space virtual network interface that connects the NUSE user-space stack with the kernel's networking subsystem. Implemented similarly to virtual Ethernet (veth), TAP, or loopback devices, it registers as a standard netdev in the kernel. Incoming packets delivered to the NUSE Driver are handed off to the NUSE System Service via shared memory (instead of being processed by the kernel's IP/TCP stack), whereas outgoing packets from the user-space stack are injected into the kernel's networking pipeline through the driver's `start_xmit()` function. This design lets NUSE seamlessly integrate with existing kernel networking infrastructure — for example, packets passed to the NUSE Driver can still go through `netif_receive_skb()` so that kernel layers (like firewall, routing, etc.) remain aware of the traffic if needed.

4.3.4 UC Channel: Userspace NIC Offloading

Achieving a zero-copy application data-path is essential for high-performance networking [246]. To achieve a flexible, high-performance packet delivery abstraction, NUSE leverages Unreliable Connection (UC) mode in RDMA (`ibverbs`) to offload networking operations to the NIC while enabling zero-copy application data transfer. Unlike Unreliable Datagram (UD), which requires a strict send/receive match and does not support zero-copy placement of payloads into application buffers, UC provides direct memory placement via UC write. This allows NUSE to deliver packet payloads directly into pre-allocated application buffers without additional copies.

While Reliable Connection (RC) mode also supports direct writes, it enforces a strong transport and reliability mechanism at the NIC level, constraining the flexibility of custom transport stacks. UC mode strikes a balance by decoupling reliability from transport, enabling NUSE to maintain precise control over transport-layer semantics while still benefiting from zero-copy RDMA acceleration. This design ensures efficient packet delivery with mini-

mal kernel involvement, allowing applications to define their own transport guarantees while achieving high throughput and low latency.

4.4 Key Implementation Challenges

Designing NUSE as a hybrid-kernel networking system introduces several non-trivial implementation challenges that impact performance, application transparency, and system efficiency. In this section, we highlight three primary challenges: balancing performance with application transparency, designing an efficient shared memory queue for inter-process communication, and handling kernel stack integration without excessive overhead.

4.4.1 Balancing Performance and Application Transparency

A fundamental challenge in NUSE is ensuring application transparency while maintaining high performance. One naive approach to eliminating memory copies is to transparently replace an application's heap with shared memory allocations. However, such a design introduces severe performance issues, particularly due to TLB shootdowns caused by frequent memory mapping changes [179].

To maximize compatibility with existing applications, NUSE cannot directly modify application memory semantics. Instead, when an application calls `send()` or `recv()`, the NUSE shim copies data into a separate buffer to maintain traditional BSD socket semantics. While this incurs an additional copy, NUSE system service can still achieve zero-copy within the stack using NIC scatter-gather or RDMA/DPDK acceleration. As long as the final transmission does not re-enter the kernel, this additional copy does not introduce extra overhead compared to conventional sockets.

However, for workloads requiring zero-copy semantics, such as large data transfers, NUSE must accommodate modern optimizations like `MSG_ZEROCOPY`. Since `MSG_ZEROCOPY` requires page-based memory registration, we implement an on-demand memory mapping mechanism: when an application sends data above a certain threshold (e.g., hundreds of KB), NUSE can opportunistically map the application's memory into the shared address space. This avoids repeated copying and ensures efficient zero-copy transfers while preserving existing

application behavior.

4.4.2 Efficient and Robust Shared Memory Queue Design

Efficient inter-process communication (IPC) is critical to NUSE's design, and shared memory queues (ring buffers) are a natural choice due to their low-latency characteristics. However, while shared memory queues are efficient for event notification (one-bit information) and batching, they introduce several challenges.

First, converting traditional function calls into shared memory queue-based interactions requires marshalling and unmarshalling arguments into fixed-sized slots, incurring variable costs depending on the complexity of function arguments. This transformation adds CPU overhead and increases implementation complexity.

Second, multithreaded applications exacerbate CPU consumption. If NUSE were to busy-poll on each application thread for scalability, CPU overhead would grow linearly with the number of threads, which is often impractical. Recent CPU advancements provide potential solutions:

- **Intel UMonitor/UMWait and User Interrupts**: Available in Sapphire Rapids and newer Intel architectures, these features allow a thread to efficiently sleep until a memory location is updated, reducing CPU wake-up costs compared to pure busy polling.
- **Intel Data Movement Library (DML)**: By leveraging Intel's Data Streaming Accelerator (DSA) ASICs, NUSE can offload memory copy operations, significantly reducing CPU usage for large data movements.

By combining busy polling for high-frequency events with low-overhead sleeping mechanisms, NUSE aims to achieve an optimal balance between low-latency responsiveness and CPU efficiency.

4.4.3 Kernel Stack Integration and SKB Conversion

While NUSE primarily operates in user space, it still needs to support scenarios where data must traverse the kernel TCP/IP stack. A key challenge is efficiently extracting packets

from the kernel and delivering them to NUSE system service without excessive overhead.

One approach is to use `io_uring` or similar mechanisms to extract packets from the NUSE driver and forward them to user space. However, `sk_buff` (SKB) structures in the Linux kernel are complex, containing multiple pointers to kernel objects. Passing SKBs directly between the NUSE driver and the NUSE system service is not feasible without translation.

To resolve this, we introduce an intermediate packet representation to serialize SKBs for safe transfer. This process may require copying packet headers (a few bytes) or even payload data, depending on the use case. While this conversion introduces some overhead, it ensures that NUSE can interact seamlessly with the kernel stack when needed.

4.5 Applicability and Limitations

NUSE provides a flexible platform for host networking research but is best suited for specific scenarios.

Applicable Scenarios. NUSE is well-suited for networking research and prototyping, allowing researchers to implement and evaluate new transport protocols and packet processing techniques with minimal changes to existing applications. By intercepting socket calls and handling networking in user space, NUSE offers a convenient testbed for evaluating novel networking concepts while maintaining compatibility with real workloads.

Beyond research, NUSE is applicable in environments that already leverage RDMA, such as cloud data centers and high-performance computing (HPC) clusters. Since RDMA-based infrastructure is increasingly prevalent in large-scale workloads like deep learning and data analytics, NUSE can serve as a platform for optimizing host networking in these domains. Applications that rely on high-throughput, low-latency communication—such as distributed deep learning—can particularly benefit from NUSE's transport-layer flexibility without requiring significant application modifications.

Non-Applicable Scenarios. NUSE is not an alternative for kernel-level networking research that requires modifications to the Linux TCP/IP stack or other core networking

subsystems. Researchers focusing on in-kernel transport optimizations, congestion control mechanisms, or low-level packet scheduling would find it necessary to work within the kernel rather than relying on a user-space framework like NUSE. Additionally, some network acceleration techniques rely on hardware-specific solutions, such as FPGAs or vendor-specific NIC offloads, which may not integrate seamlessly with NUSE's architecture.

Another limitation is that NUSE currently relies on busy polling for efficiency, which can lead to high CPU consumption. In dedicated, high-performance environments where applications can afford to dedicate CPU resources to networking, this trade-off is acceptable. However, in multi-tenant cloud environments where CPU efficiency is critical, the additional polling overhead may be prohibitive. Addressing this issue through more advanced polling mechanisms or event-driven wake-up techniques could improve NUSE's suitability for shared cloud environments.

4.6 Related Work

Extensible frameworks. Several existing platforms attempt to offer more flexible environments for networking experiments, though each has limitations relative to NUSE's objectives. CCP [196], for example, is specialized for congestion control research, focusing on transport-layer optimizations, whereas NUSE provides a broader platform for host networking experimentation. FUSE [62] is conceptually similar in elevating file system logic to userspace, yet it targets storage rather than networking, and thus does not address challenges like routing or transport-layer modifications. Snap [179] places the transport stack in userspace but does not disclose internal interfaces for modular extensions and remains closed-source. Demikernel [293] proposes a flexible datapath OS architecture designed for microsecond-scale datacenter systems, focusing on kernel-bypass devices, which is complementary to NUSE's goal of broader host networking experimentation.

Hybrid-kernel architectures. Hybrid-kernel strategies have been explored in projects such as mRPC [45] and MCCS [281], which partially bypass the kernel to achieve performance gains. Similarly, TAS [144] accelerates TCP by moving transport processing to user space,

while Snap [179] isolates host networking into a microkernel-like service. NUSE differentiates itself by providing a generalized platform that allows developers to selectively substitute or extend host network components without duplicating every aspect of the kernel stack.

Network stack architecture. Various approaches have been proposed to redesign network stack architectures to improve performance and flexibility. Kernel-bypass techniques, such as those employed in IX [29] and Demikernel [293], allow applications to directly interact with network hardware, reducing latency and overhead associated with kernel involvement. eBPF-based solutions enable safe and efficient execution of custom networking functions within the kernel, offering extensibility without compromising security. Microkernel-based network stack, like Snap [179], isolate networking components into separate user-space services, enhancing modularity and fault isolation. NUSE aligns with these efforts by facilitating user-space experimentation with a broad range of host networking ideas while maintaining compatibility with existing kernel mechanisms.

Performance optimization. Optimizing performance in networking systems has been a significant research focus, with various strategies proposed in different contexts. User-interrupts [130, 166, 162] and RDMA UC (Unreliable Connection) [160] are techniques employed to reduce CPU overhead and improve data transfer efficiency. NUSE plans to incorporate these optimizations to enhance performance, leveraging user-interrupts to minimize CPU usage during network operations and utilizing RDMA UC to achieve zero-copy of application buffer. These approaches are complementary to NUSE's design, as they address specific performance bottlenecks in host networking systems.

4.7 Summary and Future Work

NUSE introduces a hybrid-kernel networking framework that bridges the gap between user-space flexibility and kernel integration. By intercepting socket calls and leveraging RDMA-based acceleration, NUSE allows host networking experimentation without extensive kernel modifications. Preliminary results show that NUSE achieves competitive performance while maintaining application transparency, making it a viable platform for network research.

Future work will focus on improving shared memory queue efficiency to reduce polling overhead, refining zero-copy memory management to optimize large transfers, and enhancing SKB translation for better kernel interoperability. Additionally, holistic benchmarking is needed to systematically compare NUSE against other networking frameworks across diverse workloads. Addressing these challenges will further solidify NUSE as a practical tool for advancing host networking research.

5. Conclusion

In summary, this dissertation demonstrates that revisiting the interface between applications and the network can yield substantial benefits for cloud computing. We identified fundamental challenges in today's cloud datacenters: applications have little visibility into network conditions, and current mechanisms for managing communication (such as per-service RPC libraries with sidecar proxies) introduce high overhead and complexity. The root cause is the byte-level API – the narrow socket interface separating application intent from network mechanism. Our work shows that by breaking this barrier and co-designing the application-network interface, we can achieve notable improvements in both application performance and network efficiency.

We presented three complementary systems to validate this thesis. NetHint provides a practical way to break open the cloud's network black box. By sharing timely network hints from the cloud provider, NetHint allows data-intensive applications to adapt their behavior to current network topology and load. This leads to more efficient use of bandwidth and faster completion of communication-heavy operations (e.g., accelerating distributed training and data shuffles by up to 2.7×). mRPC tackles the problem of RPC management at scale by offloading RPC processing into the operating system. This approach removes the redundancy of user-level proxies and unifies control, yielding up to 2.5× lower RPC latencies and higher throughput in a representative microservice workload, as well as simplifying policy enforcement across services. Both NetHint and mRPC illustrate the power of elevating network-awareness into applications and pushing application-specific processing into the system: they significantly improve performance while making large-scale systems easier to manage.

The third piece, NUSE, broadens the scope by exploring a hybrid userspace/kernel architecture for networking. Inspired by the success of user-space filesystems (FUSE) in providing extensibility, NUSE proposes to make the host network stack modular and extensible, allowing developers to implement custom network logic in user space without kernel redesign. Unlike NetHint and mRPC, which were fully implemented and evaluated, NUSE is pre-

sented as a design proposal and prototype. Preliminary results are promising – for example, NUSE's user-space network stack achieved performance close to native RDMA for large transfers while maintaining transparency to applications – but many engineering challenges remain. I emphasize that the NUSE scheme is still in a proposal stage, representing an avenue for future work rather than a finalized system. Further research is needed to refine NUSE's components (such as optimizing its shared memory communication and zero-copy mechanisms) and to evaluate it comprehensively across diverse workloads.

Overall, the contributions of this dissertation validate the approach of application-networking co-design in cloud datacenters. By bending the traditional layers, we have enabled cooperation between applications and network infrastructure in ways that were not possible before. The outcome is a more flexible cloud platform that can cater to the needs of demanding distributed applications—improving data transfer times, reducing latency, and simplifying communication management. We showed that it is feasible to expose richer network semantics to applications (as in NetHint), and likewise feasible to embed higher-level communication services within the system (as in mRPC), all without compromising safety or isolation. These results invite cloud providers and system designers to rethink long-held abstractions.

Future research directions emerging from this work include fully realizing the NUSE vision and integrating it into production systems. This involves addressing the remaining implementation challenges and ensuring that a hybrid userspace networking stack can operate at scale and with robustness comparable to in-kernel stacks. Additionally, the general principle of co-design can be extended: for instance, designing new high-level network primitives (beyond hints and RPC) that the cloud OS could offer to applications, or exploring security and verification aspects of a more open application-network interface. There is also room to investigate how scheduling and resource allocation in datacenters could evolve if more information flows between applications and the network. By pursuing these directions, we can continue to close the gap between application intent and network operation. I believe that this line of research will shape the next generation of cloud platforms, where networking

is not just a utility beneath applications, but an intelligent, adaptive partner in distributed computing.

This dissertation is partially supported by NSF grants CNS-2238665 and CNS-2402696.

Bibliography

[1] F-Stack | High Performance Network Framework Based On DPDK. https://www.f-stack.org/, 2025.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[3] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. Towards provably performant congestion control. In *NSDI*, 2024.

[4] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Deconstructing Amazon EC2 Spot Instance Pricing. *ACM Trans. Econ. Comput.*, 2013.

[5] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[6] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

[7] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[8] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, 2010.

[9] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *SOSP*, 2001.

[10] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI*, 2014.

[11] Mahdi Arghavani, Haibo Zhang, David Eyers, and Abbas Arghavani. SUSS: Improving TCP Performance by Speeding Up Slow-Start. In *SIGCOMM*, 2024.

[12] Apache Arrow. https://arrow.apache.org/, 2022.

[13] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt: Sub-RTT Congestion Control for Ultra-Low Latency. In *NSDI 23*, 2023.

[14] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically Finding the Cause of Packet Drops. In *NSDI*, 2018.

[15] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *SIGCOMM*, 2016.

[16] Sachin Ashok, P. Brighten Godfrey, and Radhika Mittal. Leveraging Service Meshes as a New Network Layer. In *HotNets*, 2021.

[17] Jens Axboe. Efficient IO with io_uring. `https://kernel.dk/io_uring.pdf`, 2019.

[18] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *SIGCOMM*, 2015.

[19] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling ECN in Multi-Service Multi-Queue Data Centers. In *NSDI*, 2016.

[20] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A Flat Datacenter Network with Nanosecond Optical Switching. In *SIGCOMM*, 2020.

[21] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. Enabling End-Host Network Functions. In *SIGCOMM*, 2015.

[22] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[23] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*, 2013.

[24] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. Disaggregating Stateful Network Functions. In *NSDI*, 2023.

[25] Dominic Battre, Natalia Frejnik, Siddhant Goel, Odej Kao, and Daniel Warneke. Evaluation of Network Topology Inference in Opaque Compute Clouds through End-to-End Measurements. In *2011 IEEE 4th International Conference on Cloud Computing*, 2011.

[26] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

[27] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

[28] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *SIGCOMM*, 2017.

[29] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI*, 2014.

[30] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[31] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[32] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *OSDI*, 2020.

[33] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight Remote Procedure Call. *ACM Trans. Comput. Syst.*, 8(1):37–55, February 1990.

[34] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-Level Interprocess Communication for Shared Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 1991.

[35] Bincode. `https://github.com/bincode-org/bincode`, 2022.

[36] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 1984.

[37] Rajarshi Biswas, Xiaoyi Lu, and Dhabaleswar K Panda. Accelerating Tensorflow With Adaptive RDMA-Based gRPC. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, 2018.

[38] Daniel Bittman, Robert Soulé, Ethan L. Miller, Vishal Shrivastav, Pankaj Mehra, Matthew Boisvert, Avi Silberschatz, and Peter Alvaro. *Don't Let RPCs Constrain Your API*. 2021.

[39] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *SIGCOMM*, 2021.

[40] Lee Calcote and Zack Butcher. *Istio: Up and running: Using a service mesh to connect, secure, control, and observe*. O'Reilly Media, 2019.

[41] Cap'n Proto. `https://capnproto.org/`, 2022.

[42] 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.

[43] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *TOCS*, 1994.

[44] Chandra Chekuri and Kent Quanrud. Near-linear time approximation schemes for some implicit fractional packing problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 801–820. SIAM, 2017.

[45] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, and Xiaowei Yang Danyang Zhuo. Remote Procedure Call as a Managed System Service. In *NSDI*, April 2023.

[46] Jingrong Chen, Hong Zhang, Wei Zhang, Liang Luo, Jeffrey Chase, Ion Stoica, and Danyang Zhuo. NetHint: White-Box Networking for Multi-Tenant Data Centers. In *NSDI*, 2022.

[47] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *EuroSys*, 2019.

[48] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality, March 2023.

[49] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI*, 2014.

[50] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload Control for µs-scale RPCs with Breakwater. In *OSDI*, 2020.

[51] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *MLSys*, 2019.

[52] Mosharaf Chowdhury and Ion Stoica. Efficient Coflow Scheduling Without Prior Knowledge. In *SIGCOMM*, 2015.

[53] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient Coflow Scheduling with Varys. In *SIGCOMM*, 2014.

[54] Coflow-Benchmark. `https://github.com/coflow/coflow-benchmark`, 2020.

[55] Consul. `https://www.consul.io/`, 2022.

[56] Breno GS Costa, Marco Antonio Sousa Reis, Aletéia PF Araújo, and Priscila Solis. Performance and Cost Analysis Between On-Demand and Preemptive Virtual Machines. In *CLOSER*, pages 169–178, 2018.

[57] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. Virtualized Congestion Control. In *SIGCOMM*, 2016.

[58] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs. In *ASPLOS*, 2019.

[59] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *NSDI*, 2018.

[60] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[61] Mathijs Den Burger, Thilo Kielmann, and Henri E Bal. Balanced Multicasting: High-Throughput Communication for Grid Applications. In *SC*, 2005.

[62] FUSE developers. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. https://github.com/libfuse/libfuse, 2025.

[63] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. On the Impact of Packet Spraying in Data Center Networks. In *INFOCOM*, 2013.

[64] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized Task-Aware Scheduling for Data Center Networks. In *SIGCOMM*, 2014.

[65] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized Task-Aware Scheduling for Data Center Networks. In *SIGCOMM*, 2014.

[66] Adam Dunkels. Design and implementation of the lwIP TCP/IP stack. *Swedish Institute of Computer Science*, 2, 03 2001.

[67] eBPF. https://ebpf.io/, 2022.

[68] Envoy Proxy. https://www.envoyproxy.io/, 2022.

[69] Vanini Erico, Pan Rong, Alizadeh Mohammad, Taheri Parvin, and Edsall Tom. Let it Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *NSDI*, 2017.

[70] etcd. https://etcd.io/, 2022.

[71] Introducing data center fabric, the next-generation Facebook data center network. https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network, 2020.

[72] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[73] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI*, 2018.

[74] FlatBuffers. https://google.github.io/flatbuffers/, 2022.

[75] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and Flexibility in Distribution of Hot Content. In *OSDI*, 2022.

[76] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and Flexibility in Distribution of Hot Content. In *OSDI*, 2022.

[77] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-Trace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[78] S. Ben Fred, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts. Statistical Bandwidth Sharing: A Study of Congestion at Flow Level. In *SIGCOMM*, 2001.

[79] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *OSDI*, 2020.

[80] Harold N Gabow and KS Manu. Packing Algorithms for Arborescences (And Spanning Trees) In Capacitated Graphs. *Mathematical Programming*, 82(1):83–109, 1998.

[81] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *ASPLOS*, 2019.

[82] Jiaqi Gao, Nofel Yaseen, Robert MacDavid, Felipe Vieira Frujeri, Vincent Liu, Ricardo Bianchini, Ramaswamy Aditya, Xiaohang Wang, Henry Lee, David Maltz, Minlan Yu, and Behnaz Arzani. Scouts: Improving the Diagnosis Process Through Domain-Customized Incident Routing. In *SIGCOMM*, 2020.

[83] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network Aggregation for Shared Machine Learning Clusters. In A. Smola, A. Dimakis, and I. Stoica, editors, *MLSys*, 2021.

[84] Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, and Mohammad Alizadeh. JUGGLER: A Practical Reordering Resilient Network Stack for Datacenters. In *EuroSys*, 2016.

[85] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *SIGCOMM*, 2016.

[86] Soudeh Ghorbani, Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Micro Load Balancing in Data Centers with DRILL. In *HotNets*, 2015.

[87] Collective Communications Library with Various Primitives for Multi-Machine Training. https://github.com/facebookincubator/gloo, 2020.

[88] Gluster. https://www.gluster.org/, 2022.

[89] Y. Gong, B. He, and J. Zhong. Network Performance Aware MPI Collective Communication Operations in the Cloud. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):3079–3089, 2015.

[90] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.

[91] Google Cloud Network bandwidth. https://cloud.google.com/compute/docs/network-bandwidth, 2021.

[92] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open MPI: A flexible high performance MPI. In *International Conference on Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2005.

[93] gRPC. https://grpc.io/, 2022.

[94] gRPC Release Schedule. https://grpc.github.io/grpc/core/md_doc_grpc_release_schedule.html, 2022.

[95] gRPC Changelog. https://github.com/grpc/grpc/releases, 2022.

[96] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, 2019.

[97] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *SIGCOMM*, 2016.

[98] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM*, 2015.

[99] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *SIGCOMM*, 2018.

[100] Apache Hadoop. https://hadoop.apache.org/, 2020.

[101] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O . In *OSDI*, 2012.

[102] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *SIGCOMM*, 2017.

[103] HAProxy. http://www.haproxy.org/, 2022.

[104] Vipul Harsh, Sangeetha Abdu Jyothi, and P. Brighten Godfrey. Spineless Data Centers. In *HotNets*, 2020.

[105] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *MLSys*, 2019.

[106] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM*, 2015.

[107] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *SIGCOMM*, 2016.

[108] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for Virtual Private Cloud. In *SIGCOMM*, 2020.

[109] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[110] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. https://arxiv.org/abs/1704.04861, 2017.

[111] 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 *SIGCOMM*, 2014.

[112] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *SOSP*, 2021.

[113] Hydro. https://github.com/hydro-project, 2020.

[114] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *OSDI*, 2021.

[115] Istio. https://istio.io/, 2022.

[116] Rate Limit Policy in Istio. https://istio.io/latest/docs/tasks/policy-enforcement/rate-limit/, 2022.

[117] Ethan J. Jackson, Melvin Walls, Aurojit Panda, Justin Pettit, Ben Pfaff, Jarno Rajahalme, Teemu Koponen, and Scott Shenker. SoftFlow: A Middlebox Architecture for Open vSwitch. In *ATC*, 2016.

[118] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Bridging the Tenant-Provider Gap in Cloud Services. In *SOCC*, 2012.

[119] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *SIGCOMM*, 2015.

[120] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *SIGCOMM*, 2015.

[121] Pramod Jamkhedkar, Theodore Johnson, Yaron Kanza, Aman Shaikh, N.K. Shankarnarayanan, Vladislav Shkapenyuk, and Gordon Woodhull. Virtualized Network Service Topology Exploration Using Nepal. In *SIGMOD*, 2017.

[122] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *NSDI*, 2017.

[123] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W Lee. A Specialized Architecture for Object Serialization with Applications to Big Data Analytics. In *ISCA*, 2020.

[124] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable Message Latency in the Cloud. In *SIGCOMM*, New York, NY, USA, 2015. Association for Computing Machinery.

[125] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *MLSys*, 2019.

[126] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *ATC*, 2019.

[127] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications. Technical report, MSR-TR-2018-13, 2018.

[128] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[129] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[130] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. Skyloft: A General High-Efficient Scheduling Framework in User Space. In *SOSP*, 2024.

[131] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *OSDI*, 2020.

[132] Introducing JSON. `https://www.json.org/json-en.html`, 2022.

[133] Sangeetha Abdu Jyothi, Sayed Hadi Hashemi, Roy Campbell, and Brighten Godfrey. Towards An Application Objective-Aware Network Interface. In *HotCloud*, 2020.

[134] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. Finding Network Misconfigurations by Automatic Template Inference. In *NSDI*, 2020.

[135] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *NSDI*, 2019.

[136] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *USENIX ATC*, 2016.

[137] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI*, 2016.

[138] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-Scale Computer. In *ISCA*, 2015.

[139] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. A Hardware Accelerator for Protocol Buffers. In *MICRO*, 2021.

[140] Nicholas T Karonis, Bronis R De Supinski, Ian Foster, William Gropp, Ewing Lusk, and John Bresnahan. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *IPDPS*, 2000.

[141] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *NSDI*, 2018.

[142] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-Aware Load Balancing at the Virtual Edge. In *CoNEXT*, 2017.

[143] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *SOSR*, 2016.

[144] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP Acceleration as an OS Service. In *EuroSys*, 2019.

[145] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *NSDI*, 2019.

[146] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA (Network Balancing Act): A High-Performance Packet Processing Framework for Heterogeneous Processors. In *EuroSys*, 2015.

[147] Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery. In *NSDI*, 2023.

[148] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *ATC*, 2019.

[149] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 2000.

[150] Xinhao Kong, Jingrong Chen, Wei Bai, Xu Yechen, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R Lebeck, and Danyang Zhuo. Understanding RDMA Microarchitecture Resources for Performance Isolation. In *NSDI*, 2023.

[151] Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding Performance Anomalies in RDMA Subsystems. In *NSDI*, 2022.

[152] Kubernetes. `https://kubernetes.io/`, 2022.

[153] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *SIGCOMM*, 2020.

[154] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. PicNIC: Predictable Virtualized NIC. In *SIGCOMM*, 2019.

[155] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. Choreo: Network-Aware Task Placement for Cloud Applications. In *IMC*, 2013.

[156] Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. Netshare and Stochastic Netshare: Predictable Bandwidth Allocation for Data Centers. *SIGCOMM Comput. Commun. Rev.*, 2012.

[157] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs. In *ASPLOS*, 2021.

[158] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-Driven Bandwidth Guarantees in Datacenters. In *SIGCOMM*, 2014.

[159] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM*, 2016.

[160] Qiang Li, Yixiao Gao, Xiaoliang Wang, Haonan Qiu, Yanfang Le, Derui Liu, Qiao Xiang, Fei Feng, Peng Zhang, Bo Li, Jianbo Dong, Lingbo Tang, Hongqiang Harry Liu, Shaozong Liu, Weijie Li, Rui Miao, Yaohui Wu, Zhiwu Wu, Chao Han, Lei Yan, Zheng Cao, Zhongjie Wu, Chen Tian, Guihai Chen, Dennis Cai, Jinbo Wu, Jiaji Zhu, Jiesheng Wu, and Jiwu Shu. Flor: An Open High Performance RDMA Framework Over Heterogeneous RNICs. In *OSDI*, 2023.

[161] Tianxi Li, Haiyang Shi, and Xiaoyi Lu. HatRPC: Hint-Accelerated Thrift RPC over RDMA. In *SC*, 2021.

[162] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G. Edward Suh, Kostis Kaffes, and Christina Delimitrou. LibPreemptible: Enabling Fast, Adaptive, and Hardware-Assisted User-Space Scheduling. In *HPCA*, 2024.

[163] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *SIGCOMM*, 2019.

[164] Zhaogeng Li, Ning Liu, and Jiaoren Wu. Toward a Production-Ready General-Purpose RDMA-Enabled RPC. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, 2019.

[165] Libfabric. `https://ofiwg.github.io/libfabric/`, 2022.

[166] Jiazhen Lin, Youmin Chen, Shiwei Gao, and Youyou Lu. Fast Core Scheduling with Userspace Process Abstraction. In *SOSP*, 2024.

[167] Linkerd. `https://linkerd.io/`, 2022.

[168] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit Switching Under the Radar with REACToR. In *NSDI*, 2014.

[169] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A Fault-Tolerant Engineered Network. In *NSDI*, 2013.

[170] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *SIGCOMM*, 2019.

[171] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM*, 2016.

[172] Rober Love. *Linux System Programming*. O'Reilly Media, 2007.

[173] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[174] Lpsolve. `http://web.mit.edu/lpsolve_v5520/doc/index.htm`, 2020.

[175] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter Hub: a Rack-Scale Parameter Server for Distributed Deep Neural Network Training. *SoCC*, 2018.

[176] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. PLink: Discovering and Exploiting Locality for Accelerated Distributed Training on the Public Cloud. In *MLSys*, 2020.

[177] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*, 2012.

[178] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI*, 2014.

[179] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A Microkernel Approach to Host Networking. In *SOSP*, 2019.

[180] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *NSDI*, 2022.

[181] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. RotorNet: A Scalable, Low-Complexity, Optical Datacenter Network. In *SIGCOMM*, 2017.

[182] Memcached. `https://memcached.org/`, 2022.

[183] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *SIGCOMM*, 2017.

[184] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High Velocity Kernel File Systems with Bento. In *FAST*, 2021.

[185] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*, 2015.

[186] Michael Mitzenmacher. How Useful Is Old Information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20, 2000.

[187] Jeffrey C. Mogul and John Wilkes. Nines Are Not Enough: Meaningful Metrics for Clouds. In *HotOS*, 2019.

[188] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *SOSP*, 2021.

[189] MongoDB. `https://www.mongodb.com`, 2022.

[190] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *SIGCOMM*, 2018.

[191] Fabrizio Montesi and Janine Weber. Circuit Breakers, Discovery, and API Gateways in Microservices. *arXiv preprint arXiv:1609.05830*, 2016.

[192] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *NSDI*, 2020.

[193] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*, 2018.

[194] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*, 2018.

[195] MessagePack. `https://msgpack.org/index.html`, 2022.

[196] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring endpoint congestion control. In *SIGCOMM*, 2018.

[197] The NVIDIA Collective Communication Library (NCCL). `https://developer.nvidia.com/nccl`, 2020.

[198] Nginx. `https://www.nginx.com/`, 2022.

[199] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making Network Stack Part of the Virtualized Infrastructure. In *USENIX ATC*, 2020.

[200] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai Network: A Platform for High-Performance Internet Applications. *SIGOPS Oper. Syst. Rev.*, 2010.

[201] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless Datacenter Load-balancing with Beamer. In *NSDI*, 2018.

[202] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *ATC*, 2014.

[203] OpenAI. GPT-4 Technical Report, 2024.

[204] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *NSDI*, 2019.

[205] John Ousterhout. A Linux Kernel Implementation of the Homa Transport Protocol. In *ATC*, 2021.

[206] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *SOSP*, 2015.

[207] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *OSDI*, 2016.

[208] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*. 2019.

[209] Pitch Patarasuk and Xin Yuan. Bandwidth Efficient All-reduce Operation on Tree Topologies. In *IPDPS*, 2007.

[210] Y Peng, Y Zhu, Y Chen, Y Bao, B Yi, C Lan, C Wu, and C Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *SOSP*, 2019.

[211] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *EuroSys*, 2018.

[212] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *OSDI*, 2014.

[213] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[214] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. MigrOS: Transparent Live-Migration Support for Containerised RDMA Applications. In *ATC*, 2021.

[215] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze: Stateful Packet Processing in Hardware. In *NSDI*, 2019.

[216] Lucian Popa, Ali Ghodsi, and Ion Stoica. HTTP as the Narrow Waist of the Future Internet. Hotnets-IX, 2010.

[217] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[218] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*, 2013.

[219] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *ASPLOS*, 2020.

[220] Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. Cerebros: Evading the RPC Tax in Datacenters. In *MICRO*, 2021.

[221] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter Evolving: Transforming Google's Datacenter Network via Optical Circuit Switches and Software-Defined Networking. In *SIGCOMM*, 2022.

[222] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *SOSP*, 2017.

[223] Protocol Buffers. `https://developers.google.com/protocol-buffers`, 2022.

[224] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low Latency Geo-Distributed Data Analytics. In *SIGCOMM*, 2015.

[225] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for End-Host Rate Limiting. In *NSDI*.

[226] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *SIGCOMM*, 2007.

[227] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather. In *HotOS*, 2021.

[228] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather. In *HotOS*, HotOS '21, 2021.

[229] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.

[230] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, number 130-136. Citeseer, 2015.

[231] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. ReDMArk: Bypassing RDMA security mechanisms. In *USENIX Security*, 2021.

[232] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*, 2015.

[233] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. Passive Realtime Datacenter Fault Detection and Localization. In *NSDI*, 2017.

[234] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We Need Kernel Interposition over the Network Dataplane. In *HotOS*, 2021.

[235] Russel Sandberg. The Sun Network File System: Design, Implementation and Experience. In *USENIX Summer ATC*, 1986.

[236] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. Technical report, KAUST, Feb 2019. http://hdl.handle.net/10754/631179.

[237] S. Savage, T. Anderson, Amit Aggarwal, David Becker, N. Cardwell, A. Collins, Eric Hoffman, John Snell, Amin Vahdat, G. Voelker, and J. Zahorjan. Detour: Informed Internet Routing and Transport. *IEEE Micro*, 19:50–59, 1999.

[238] Brandon Schlinker, Radhika Niranjan Mysore, Sean Smith, Jeffrey C. Mogul, Amin Vahdat, Minlan Yu, Ethan Katz-Bassett, and Michael Rubin. Condor: Better Topologies Through Declarative Design. In *SIGCOMM*, 2015.

[239] Mike Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transaction on Computer Systems*, February 1990.

[240] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI*, 2018.

[241] Haiying Shen, Ankur Sarker, Lei Yu, and Feng Deng. Probabilistic Network-Aware Task Placement for MapReduce Scheduling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 241–250. IEEE, 2016.

[242] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[243] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: Productionizing an RMA-Based Distributed Caching System. In *SIGCOMM*, 2021.

[244] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *SIGCOMM*, 2020.

[245] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.

[246] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, Raghu Raja, Daniel Walton, Rachit Agarwal, Shrijeet Mukherjee, and Christos Kozyrakis. High-throughput and Flexible Host Networking for Accelerated Computing. In *OSDI*, 2024.

[247] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. *Facebook white paper*, 5(8):127, 2007.

[248] Snabb: Simple and fast packet networking. https://github.com/snabbco/snabb, 2022.

[249] Marco Spuri and Giorgio C. Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Real-Time Systems Symposium*, pages 2–11, 1994.

[250] Brent Stephens, Aditya Akella, and Michael M. Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *NSDI*, 2019.

[251] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. Titan: Fair Packet Scheduling for Commodity Multiqueue NICs. In *USENIX ATC*, 2017.

[252] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. DaRPC: Data Center RPC. In *SoCC*, 2014.

[253] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is Faster than Server-Bypass with RDMA. In *EuroSys*, 2017.

[254] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. The NEBULA RPC-Optimized Architecture. In *ISCA*, 2020.

[255] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *CVPR*, 2016.

[256] Mingxing Tan and Quoc Le. Efficientnet: Rethinking Model Scaling for Convolutional Neural Networks. In *ICML*, 2019.

[257] Puqi Perry Tang and Tsung-Yuan Charles Tai. Network Traffic Characterization Using Token Bucket Model. In *INFOCOM*, 1999.

[258] Indu Thangakrishnan, Derya Cavdar, Can Karakus, Piyush Ghai, Yauheni Selivonchyk, and Cory Pruce. Herring: Rethinking the Parameter Server at Scale for the Cloud. In *SC*, 2020.

[259] Apache Thrift. https://thrift.apache.org/, 2022.

[260] Tonic. https://github.com/hyperium/tonic, 2022.

[261] Torchelastic. https://pytorch.org/elastic/0.2.2/index.html, 2020.

[262] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *SIGMOD*, 2014.

[263] Hugo Touvron et al. Llama 2: Open Foundation and Fine-Tuned Chat Models, 2023.

[264] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *SOSP*, 2017.

[265] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is Big Data Performance Reproducible in Modern Cloud Networks? In *NSDI*, 2020.

[266] Amin Vadhat. Coming of Age in the Fifth Epoch of Distributed Computing: The Power of Sustained Exponential Growth, 2020. Amin Vahdat - SIGCOMM Lifetime Achievement Award 2020 Keynote.

[267] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. Exploiting Spot and Burstable Instances for Improving the Cost-Efficacy of In-Memory Caches on the Public Cloud. In *EuroSys*, 2017.

[268] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and Generic Collectives for Distributed ML. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *MLSys*, 2020.

[269] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards Domain-Specific Network Transport for Distributed DNN Training. In *NSDI*, 2024.

[270] R. Wang, J. A. Wickboldt, R. P. Esteves, L. Shi, B. Jennings, and L. Z. Granville. Using Empirical Estimates of Effective Bandwidth in Network-Aware Placement of Virtual Machines in Datacenters. *IEEE Transactions on Network and Service Management*, 13(2):267–280, 2016.

[271] Shibo Wang, Shusen Yang, Xiao Kong, Chenglei Wu, Longwei Jiang, Chenren Xu, Cong Zhao, Xuesong Yang, Jianjun Xiao, Xin Liu, Changxi Zheng, Jing Wang, and Honghao Liu. Pudica: Toward Near-Zero Queuing Delay in Congestion Control for Cloud Gaming. In *NSDI*, 2024.

[272] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A Distributed Futures System for Fine-Grained Tasks. In *NSDI*, 2021.

[273] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, T. S. Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, Robust, and Practical Datacenter CC via Deployable INT. In *NSDI*, 2023.

[274] Andy Warfield. Building and Operating a Pretty Big Storage System (My Adventures in Amazon S3). In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, 2023.

[275] Matt Welsh and David Culler. Adaptive Overload Control for Busy Internet Servers. In *4th USENIX Symposium on Internet Technologies and Systems (USITS 03)*, 2003.

[276] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI*, 2011.

[277] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI*, 2011.

[278] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards Zero-Copy Serialization. In *HotOS*, 2021.

[279] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Gener. Comput. Syst.*, 15(5–6):757–768, October 1999.

[280] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated Bug Removal for Software-Defined Networks. In *NSDI*, 2017.

[281] Yongji Wu, Yechen Xu, Jingrong Chen, Zhaodong Wang, Ying Zhang, Matthew Lentz, and Danyang Zhuo. MCCS: A Service-based Approach to Collective Communication for Multi-Tenant Cloud. In *SIGCOMM*, 2024.

[282] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI*, 2018.

[283] Jiali Xing, Henri Maxime Demoulin, Konstantinos Kallas, and Benjamin C. Lee. Charon: A Framework for Microservice Overload Control. In *HotNets*, 2021.

[284] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, Ziyang Yang, Hongyi Liu, and Ang Chen. Bedrock: Programmable Network Support for Secure RDMA Systems. In *USENIX Security*, 2022.

[285] Praveen Yalagandula and Mike Dahlin. A Scalable Distributed Information Management System. *ACM SIGCOMM Computer Communication Review*, 34(4):379–390, 2004.

[286] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. Twenty Years After: Hierarchical Core-Stateless Fair Queueing. In *NSDI*, 2021.

[287] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.

[288] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[289] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 2016.

[290] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *ATC*, 2017.

[291] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark. In *SIGCOMM*, 2016.

[292] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient Datacenter Load Balancing in the Wild. In *SIGCOMM*, 2017.

[293] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *SOSP*, 2021.

[294] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. Gallium: Automated Software Middlebox Offloading to Programmable Switches. In *SIGCOMM*, 2020.

[295] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-Resolution Measurement of Data Center Microbursts. In *IMC*, 2017.

[296] Yiran Zhang, Qingkai Meng, Chaolei Hu, and Fengyuan Ren. Revisiting Congestion Control for Lossless Ethernet. In *NSDI*, 2024.

[297] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *NSDI*, 2022.

[298] Yangming Zhao, Kai Chen, Wei Bai, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. RAPIER: Integrating Routing and Scheduling for Coflow-aware Data Center Networks. In *INFOCOM*, 2015.

[299] Yangming Zhao, Chen Tian, Jingyuan Fan, Tong Guan, and Chunming Qiao. RPC: Joint Online Reducer Placement and Coflow Bandwidth Scheduling for Clusters. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018.

[300] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload Control for Scaling WeChat Microservices. In *SoCC*, 2018.

[301] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow Event Telemetry on Programmable Data Plane. 2020.

[302] Xiangfeng Zhu, Weixin Deng, Banruo Liu, Jingrong Chen, Yongji Wu, Thomas Anderson, Arvind Krishnamurthy, Ratul Mahajan, and Danyang Zhuo. Application Defined Networks. In *HotNets*, 2023.

[303] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng-Ju He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting Service Mesh Overheads. *ArXiv*, abs/2207.00592, 2022.

[304] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015.

[305] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *SIGCOMM*, 2015.

[306] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems. In *SIGCOMM*, 2021.

[307] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and Mitigating Packet Corruption in Data Center Networks. In *SIGCOMM*, 2017.

[308] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Hang Guan, Arvind Krishnamurthy, and Thomas Anderson. RAIL: A Case for Redundant Arrays of Inexpensive Links in Data Center Networks. In *NSDI*, 2017.

[309] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. Slim: OS Kernel Support for a Low-Overhead Container Overlay Network. In *NSDI*, 2019.

Biography

Jingrong Chen is a Ph.D. candidate at Duke University, advised by Professor Danyang Zhuo, since August 2020. His research spans from layer-4 to layer-7 networking, with a focus on performance and manageability in end-host networking in cloud datacenters. He has built software infrastructures that enable fast and manageable network access for distributed applications and microservices, by exploring alternative software designs. Before Duke, he received his MPhil degree at HKUST and BS degree at Fudan University.