UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
High-Performance Cloud Computing Frameworks for Serverless Computing and 5G Systems

Permalink
https://escholarship.org/uc/item/6z07g3qp

Author
Qi, Shixiong

Publication Date
2024

Copyright Information
This work is made available under the terms of a Creative Commons Attribution-ShareAlike License, available at https://creativecommons.org/licenses/by-sa/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

High-Performance Cloud Computing Frameworks for Serverless Computing and 5G Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Shixiong Qi

June 2024

Dissertation Committee:

Dr. K. K. Ramakrishnan, Chairperson
Dr. Jiasi Chen
Dr. Nael Abu-Ghazaleh
Dr. Rajiv Gupta
Dr. Zhaowei Tan

Copyright by
Shixiong Qi
2024

The Dissertation of Shixiong Qi is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I would like to take this opportunity to thank everyone who helped me throughout this journey and complete this dissertation.

First, I would like to express my deepest gratitude to my advisor, Prof. K. K. Ramakrishnan for his excellence guidance and support during my PhD. China has an old saying: "*Thousand-mile horses often exist, but Bo-Le does not always exist.*" Bo-Le is the person who has good judgments to spot thousand-mile horses. Prof. K. K. is my Bo-Le, who recognized me when I had nothing to show for him and improved me to be a good researcher. The completion of this dissertation would not have been possible without his mentorship and invaluable insights. I would like to thank all the members of my dissertation committee: Prof. Rajiv Gupta, Prof. Jiasi Chen, Prof. Nael Abu-Ghazaleh, Prof. Zhaowei Tan, for their insightful comments. I would also like to thank Prof. Michalis Faloutsos for his helpful comments during his service on my PhD candidacy committee.

I would like to extend a special thank you to Prof. Jiasi Chen, Prof. Nael Abu-Ghazaleh, and Prof. Timothy Wood for providing letters of recommendation during my academic job search. I am very grateful to Prof. Timothy Wood for his guidance in preparing me for my academic job interview. I would also like to sincerely thank Prof. Emiliano De Cristofaro for his advice in preparing the job application documents.

I would like to thank all the researchers that I was fortunate enough to collaborate with in various projects during my PhD: Prof. Jyh-Cheng Chen (National Yang Ming Chiao Tung University), Prof. Timothy Wood (George Washington University), Dr. Myungjin Lee (Cisco Research), Prof. Sameer G. Kulkarni (Indian Institute of Technology Gandhinagar).

I would like to thank my industry mentors and colleagues for their support and patience throughout my internships: Dr. Puneet Sharma, Dr. Diman Zad Tootaghaj, and Dr. Hardik Soni (Hewlett Packard Labs), Dr. Poornima Lalwaney (Intel).

I would like to thank all excellent students I worked with during my PhD: Mr. Federico Parola (Politecnico di Torino), Mr. Ziteng Zeng, Mr. Leslie Monis, and Mr. Anvaya Narappa (University of California, Riverside), Mr. Han-Sing Tsai, Mr. Yu-Sheng Liu, Mr. Hao-Tse Chu, and Mr. Chia-An Lee (National Yang Ming Chiao Tung University).

I would like to thank past and current members of my lab who helped me in various stages of my PhD. I am especially thankful to Victor G. Hill, whose technical help was crucial for all our research.

Finally, I would like to thank Qi's family. Without their unwavering support, this journey would not have been possible. I am deeply grateful to my entire family. It was a generational effort and not just my struggle alone. I am grateful to my grandfather (Jia-Zheng Qi, passing in 2014) for shaping my morals and my mother (Hong-Bin Qi, passing in 2010) for nurturing my behavior, to my stepfather (Dong-Min Wang) for his tremendous financial support, and to my mother's sibling sisters (Hong-Xia Qi, Hong-Ling Qi, Hong-Yan Qi) for their care and support for more than decades.

This dissertation includes content published in the following proceedings and journals (those marked with '**' are described thoroughly while those marked with '*' are mentioned briefly in this dissertation):

1. Shixiong Qi, K. K. Ramakrishnan, Jyh-Cheng Chen, "L26GC: Evolving the Low Latency Core for Future Cellular Networks," IEEE Internet Computing, 2024. **

2. Shixiong Qi, K. K. Ramakrishnan, Myungjin Lee, “LIFL: A Lightweight, Event-driven Serverless Platform for Federated Learning,” The Seventh Conference on Machine Learning and Systems (MLSys 2024). **

3. Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, K. K. Ramakrishnan, “SPRIGHT: High-performance eBPF-based Event-driven, Shared-memory Processing for Serverless Computing,” IEEE/ACM Transactions On Networking, 2024. **

4. Yu-Sheng Liu, Shixiong Qi, Po-Yi Lin, Han-Sing Tsai, K. K. Ramakrishnan, Jyh-Cheng Chen, “L25GC+: An Improved, 3GPP-compliant 5G Core for Low-latency Control Plane Operations,” The 12th IEEE International Conference on Cloud Networking (IEEE CloudNet 2023). **

5. Shixiong Qi, Han-Sing Tsai, Yu-Sheng Liu, K. K. Ramakrishnan, Jyh-Cheng Chen, “X-IO: A High-performance Unified I/O Interface using Lock-free Shared Memory Processing,” The 9th IEEE International Conference on Network Softwarization (IEEE NetSoft 2023). **

6. Shixiong Qi, Ziteng Zeng, Leslie Monis, and K. K. Ramakrishnan, “MiddleNet: A Unified, High-Performance NFV and Middlebox Framework with eBPF and DPDK,” IEEE Transactions on Network and Service Management, 2023. *

7. Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, K. K. Ramakrishnan, “ SPRIGHT: Extracting the Server from Serverless Computing! High-Performance eBPF-based Event-driven, Shared-Memory Processing,” ACM SIGCOMM 2022. **

8. Vivek Jain, Hao-Tse Chu, Shixiong Qi, Chia-An Lee, Hung-Cheng Chang, Cheng-

Ying Hsieh, K. K. Ramakrishnan, Jyh-Cheng Chen, "L25GC: A Low Latency 5G Core Network based on High-Performance NFV Platforms," ACM SIGCOMM 2022. *

9. Ziteng Zeng, Leslie Monis, Shixiong Qi, K. K. Ramakrishnan, "MiddleNet: A High-Performance, Lightweight, Unified NFV and Middlebox Framework," The 8th IEEE International Conference on Network Softwarization (IEEE NetSoft 2022). *

10. Vivek Jain, Sourav Panda, Shixiong Qi, K. K. Ramakrishnan, "Evolving to 6G: Improving the Cellular Core to Lower Control and Data Plane Latency," The 1st International Conference on 6G Networking (6GNet 2022). *

11. Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G Kulkarni, Dan Li, Jinho Hwang, K. K. Ramakrishnan, Timothy Wood, "Mu: An Efficient, Fair and Responsive Serverless Framework for Resource-Constrained Edge Clouds," 2021 ACM Symposium on Cloud Computing (SoCC'21). *

12. Ian-Chin Wang, Shixiong Qi, Elizabeth Liri and K. K. Ramakrishnan, "Towards a Proactive Lightweight Serverless Edge Cloud for Internet-of-Things Applications," The 15th International Conference on Networking, Architecture, and Storage (NAS 2021). *

13. Vivek Jain, Shixiong Qi and K. K. Ramakrishnan, "Fast Function Instantiation with Alternate Virtualization Approaches," 2021 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN 2021). *

14. Shixiong Qi, Sameer G. Kulkarni, and K. K. Ramakrishnan, "Assessing Container

Network Interface Plugins: Functionality, Performance, and Scalability," IEEE Transactions on Network and Service Management, 2020. **

15. Shixiong Qi, Sameer G. Kulkarni, and K. K. Ramakrishnan, "Understanding Container Network Interface Plugins: Design Considerations and Performance," 2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN 2020). **

To the family of Qi for all the support.

ABSTRACT OF THE DISSERTATION

High-Performance Cloud Computing Frameworks for Serverless Computing and 5G Systems

by

Shixiong Qi

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2024
Dr. K. K. Ramakrishnan, Chairperson

Cloud computing has fundamentally transformed the landscape of computing by leveraging the widespread accessibility of network connectivity. In this space, serverless computing platforms have emerged as key facilitators, promising cost-effective cloud computing capabilities for users. However, existing serverless platforms face several challenges, particularly in the data plane they use for communication between serverless functions. These data plane mechanisms conflict with the event-driven philosophy of serverless computing. They do not help meet the desired low-latency requirements crucial for a variety of services using the cloud. For example, 5G cellular networks are transitioning to a cloud-native design, adopting a Service-based Architecture, to simplify the development and deployment of 5G systems in the cloud. However, this comes with trade-offs, as it may limit performance and hinder the execution of low-latency operations in 5G cellular core network (5GC).

Our first contribution focuses on assessing the overhead of container network interface, the essential networking component in the serverless data plane. Our investigation revealed that current serverless platforms treat inter-function networking as though each

function were isolated by a network link in the kernel, a methodology fraught with overhead. We recognize the imperative of facilitating seamless communication within nodes via shared memory processing and design SPRIGHT. SPRIGHT is the first serverless framework that leverages the extended Berkeley Packet Filter (eBPF), an in-kernel event-driven networking subsystem, to supplant long-running stateful components typically used in current serverless platforms. We also optimize the placement engine in the serverless control plane to account for heterogeneity and fairness across competing functions, ensuring overall resource efficiency, and minimizing resource fragmentation.

We then show how an improved serverless design (called LIFL) can be suitable to support fast and efficient model aggregation in Federated Learning (FL), which typically involves varying numbers of heterogeneous clients locally training machine learning models. LIFL adopts the streamlined data plane from SPRIGHT. We further introduce locality-aware placement in LIFL to maximize the benefits of shared memory processing. LIFL precisely scales and carefully reuses server resources for hierarchical aggregation to achieve the highest degree of parallelism while minimizing aggregation time and resource consumption. Our open-source implementation of LIFL achieves significant improvement in resource efficiency and aggregation speed for supporting FL at scale, compared to existing serverful as well as serverless FL systems.

Our last contribution introduces L25GC+, a 3GPP-compliant 5G Core designed to support low-latency control plane operations and high-performance user plane packet transmissions. In the control plane, L25GC+ leverages the 3GPP-specified Service-Based Interface (SBI) for exchanging control events between 5G functions. L25GC+ addresses this

by redesigning the SBI using shared memory processing instead of kernel networking. We augment asynchronous shared memory I/O with synchronous data exchange primitives and stateful processing to differentiate across user sessions. The optimized design seamlessly integrates with the 3GPP-compliant SBI, demonstrating improved latency, scalability, and user experience on commercial 5G testbeds. In the 5GC data plane, packet classification becomes expensive with a growing number of user sessions. We devise faster packet classification approaches in the 5GC data plane, ensuring high data plane performance, even with an increasing number of user sessions.

Contents

List of Figures **xvi**

List of Tables **xx**

1 Introduction **1**

2 Related Work **11**
2.1 Container Networking 11
2.2 Function Placement in a Serverless Cloud 13
2.3 Optimizing Serverless Computing Data Plane 15
2.4 ‘Cold Start’ in Serverless Computing 17
2.5 Federated Learning Optimization 17
2.6 Optimizing 5G Core Control Plane 18

3 Assessing Container Network Interface: Functionality, Performance, and Scalability **20**
3.1 Introduction 20
3.2 Background 24
3.2.1 Container Network Interface 25
3.2.2 Linux Network Namespace and Kubernetes Namespace 26
3.2.3 Kubernetes Networking Model 27
3.2.4 Kubernetes Network Policy 28
3.3 Qualitative Comparison on CNI Plugins 29
3.3.1 Description of each CNI plugin 34
3.3.2 Iptables Comparison 40
3.3.3 Summary of Qualitative Comparison 43
3.3.4 Additional Feature Considerations 44
3.4 Quantitative Evaluation and Analysis 45
3.4.1 Experimental Setup 45
3.4.2 Intra-Host Performance 46
3.4.3 Inter-Host Performance 50
3.4.4 Performance for larger-scale configurations 58

3.4.5 Impact of CNI on typical HTTP workload 62
3.4.6 Iptables Evaluation 64
3.4.7 Pod Creation Time Analysis 67
3.5 Characteristics for an Ideal CNI 68
3.6 Conclusion 69

4 An Efficient and Fair Algorithm for Serverless Function Placement 71
4.1 Introduction 71
4.2 Background 75
4.2.1 Kubernetes Scheduler (Placement Engine) 75
4.2.2 Knative Serverless Platform 76
4.3 Placement Engine Design 77
4.3.1 Optimization Model and Metrics 78
4.3.2 Heuristic algorithms 81
4.4 Placement engine evaluation 83
4.4.1 Simulation 83
4.4.2 Evaluation of Mu's Placement Engine with Real Workloads 85
4.5 Conclusion 90

5 High-performance and Efficient Serverless Computing using Event-driven Shared Memory Processing 91
5.1 Introduction 91
5.2 Background and Challenges 95
5.3 System Design of SPRIGHT 101
5.3.1 Overview of SPRIGHT 101
5.3.2 Event-driven processing 105
5.3.3 Shared memory communication within function chains 112
5.3.4 Direct Function Routing within a function chain 119
5.3.5 Function startup in SPRIGHT 120
5.3.6 Event-driven protocol adaptation 121
5.3.7 Security domains in SPRIGHT 122
5.3.8 Discussion 123
5.4 Evaluation & Analysis 125
5.4.1 Experiment Setup 125
5.4.2 Performance with Realistic Workloads 130
5.4.3 Startup latency comparison (SPRIGHT vs. Knative) 137
5.5 Conclusion 141

6 LIFL: A Lightweight, Event-driven Serverless Platform for Federated Learning 143
6.1 Introduction 143
6.2 Background and Challenges 147
6.2.1 Basics of Federated Learning 147
6.2.2 Anatomy of Systems for Federated Learning 148
6.2.3 Motivation and Challenges for Serverless FL 150

6.3 LIFL Overview 152
6.4 Optimizing the Serverless Data-Plane in LIFL 155
6.4.1 Shared Memory for Hierarchical Aggregation 155
6.4.2 In-place Message Queuing 157
6.4.3 eBPF-based Sidecar 159
6.4.4 Direct Routing with Hierarchical Aggregation 159
6.4.5 Abstraction for fine-grained control 162
6.4.6 Step-based Processing Model 163
6.5 LIFL's Control Plane Design 163
6.5.1 Locality-aware Placement and Load Balancing 163
6.5.2 Planning the Hierarchy for Aggregation 165
6.5.3 Opportunistic Reuse of Aggregator Instances 167
6.5.4 Eager aggregation in LIFL 167
6.6 Evaluation & Analysis 168
6.6.1 Microbenchmark Analysis 170
6.6.2 FL Workloads Setup 177
6.6.3 Putting It All Together 179
6.7 Conclusion 182

7 High-performance, 3GPP-compliant 5G and Beyond Systems 183
7.1 Introduction 183
7.2 Background and Motivation 189
7.2.1 Evolving the Architecture of the Cellular Core Network 189
7.2.2 Emerging 5G application needs 190
7.2.3 Limitations of 3GPP Cellular Core 192
7.3 Improved Control Plane Design in $\texttt{L}^2\texttt{5GC+}$ 194
7.3.1 Overview of $\texttt{L}^2\texttt{5GC+}$'s SBI 194
7.3.2 Shared memory management in $\texttt{L}^2\texttt{5GC+}$ 197
7.3.3 Message descriptor delivery in $\texttt{L}^2\texttt{5GC+}$ 198
7.3.4 Building the SBI over shared memory: detailed design 198
7.3.5 Concurrent user session support 202
7.3.6 Deployment strategy of $\texttt{L}^2\texttt{5GC+}$ 204
7.4 Fast and Scalable Packet Classification in $\texttt{L}^2\texttt{5GC+}$'s User Plane Function . . 205
7.4.1 Partitioning-based Versus Decision-tree-based Packet Classifiers . . . 205
7.4.2 PDR Lookup and Update in $\texttt{L}^2\texttt{5GC+}$ 207
7.5 Evaluation 208
7.5.1 Control Plane Evaluation with ***commercial*** UEs and RAN 208
7.5.2 Control Plane Evaluation with ***simulated*** UEs and RAN 212
7.5.3 Data Plane Evaluation 213
7.6 Conclusion 216

8 Conclusions 217

Bibliography 221

List of Figures

3.1 Kubernetes Cluster and the Role of CNI. 21
3.2 CNI Plugin: Interfacing w/Kubernetes & Namespaces 24
3.3 Network model for CNIs operating at Layer-3 in overlay mode (*e.g.,* Calico, Cilium) 31
3.4 Network model for CNIs operating in Layer-3 in underlay mode (*e.g.,* Calico, Cilium) 32
3.5 Network model for CNIs operating in both Layer-2 & Layer-3 in overlay mode (*e.g.,* Flannel, Weave, Kube-router) 32
3.6 Network model for CNIs operating in both Layer-2 & Layer-3 in underlay mode (*e.g.,* Flannel, Kube-router) 33
3.7 Iptables chain processing and hook points [84]. 41
3.8 Intra-Host Throughput and Latency 48
3.9 Overhead Breakdown for Intra-host scenario. 48
3.10 Inter-Host Throughput and Latency. (a) and (b) are with Mellanox ConnectX-4 25Gb NIC. (c) and (d) are with Intel X520-DA2 10Gb NIC. 52
3.11 Overhead Breakdown for Inter-host Scenario: CPU overhead from both sender and receiver hosts 53
3.12 Intra-Host Performance with increasing # connections 57
3.13 Inter-Host Performance with increasing # background connections; CPU and memory overheads 61
3.14 HTTP Performance for inter-host communication 63
3.15 Pod Creation Latency with different CNIs. 66

4.1 The scheduling model of Kubernetes 74
4.2 The working model of Knative platform 77
4.3 Mu Overview. 78
4.4 An example on integrating the degree of CPU unfairness over 30 intervals . 80
4.5 Fairness and efficiency comparison (simulation) 83
4.6 Response time CDF for 3 frameworks for Workload 1 (left); Workload 2 (right; only partial CDF for Concurrency) 85
4.7 Time series of Response Time for Mu, RPS, and Concurrency (Top: Workload 1; Bottom: Workload 2) 86

4.8 Time series of Pod counts for Mu (left), RPS (middle), and Concurrency (right) 88

5.1 Networking processing involved in a typical serverless function chain. 95
5.2 Performance and overhead breakdown of different sidecar proxies. 98
5.3 The overall architecture of SPRIGHT. 101
5.4 Event-driven EPROXY & SPROXY, shared memory, and DFR within a chain: (①) the SPRIGHT gateway invokes the head function of chain; (②) the head function calls the next function bypassing the SPRIGHT gateway. 106
5.5 Initialization of SPROXY: `SKMSG` program, sockmap. 109
5.6 Dataplane acceleration using eBPF XDP/TC hooks 111
5.7 (Left) Performance impact of TC/XDP redirect: RPS and latency; (Right) CPU overhead breakdown of receiver side kernel stacks: with TC/XDP acceleration (w/ acc.) & without TC/XDP acceleration (w/o acc.). 113
5.8 Comparison between polling-based (D-SPRI.) and event-driven (S-SPRI.) shared memory processing with 1 gateway pod and 2 function pods. Kn: Knative; QPs: Sidecars; SFs: serverless functions; GW: gateway. All results show the 99% confidence interval. 117
5.9 Serverless function chains setup. The Parking workload has two function chain invocation sequences: (Chain-1) ①→②→③→⑤→④; (Chain-2) ①→②→④ 127
5.10 RPS for online boutique: {Knative, gRPC} at 5K & {D-SPRIGHT, S-SPRIGHT (overlap)} at 25K concurrency. 128
5.11 Online boutique. Top row: Knative, 5K concurrency. Mid. row: gRPC, 5K concurrency. Bottom row: {D-SPRIGHT (D), S-SPRIGHT (S)}, 25K concurrency. (Left col.) Response time CDF for 6 different function chains; (Mid. col.) Time series of response time, function chains; (Right col.) CPU usage time series, gateway (GW), function chains (*fn*), sidecar proxy (Knative) 129
5.12 Time series of response time, and CPU utilization for motion detection workload - 1-hour long experiments. 134
5.13 Parking image detection & charging: (a) Time series of response time of function chains; (b) Time series of aggregate CPU for function chains, sidecar proxy (Knative). 136
5.14 (Left) Latency comparison of a single function pod startup between SPRIGHT and Knative; and (Right) Latency of multiple function pods startup in SPRIGHT (startup in parallel VS. startup in sequence). For function's index, refer to Fig. 5.9. 139

6.1 Synchronous FL with different aggregation timing ("Eager" and "Lazy") [87, 134]. 146
6.2 Generic architectures for FL systems: (a) Serverful FL systems [87, 126]; (b) Serverless FL systems [135, 133, 94]. Note that for simplicity, we skip the hierarchy in the diagram (b). 149
6.3 The overall architecture of LIFL. 153

6.4 Impact of data plane performance on hierarchical aggregation. (upper fig.:) No hierarchy(NH); (lower fig.:) With hierarchy(WH). Top: top aggregator; LF: leaf aggregator. "Network" denotes the data transfer tasks of model updates; "Agg." denotes the aggregation tasks; "Eval." denotes the evaluation tasks. 155
6.5 Message queuing solutions. 158
6.6 Intra-/inter-node direct routing within hierarchical aggregation. 160
6.7 Step-based processing model. 162
6.8 Control plane orchestration in LIFL: The autoscaler periodically re-plans the hierarchy based on the arrival rate of each worker node. The LIFL coordinator applies reusing of aggregators. 165
6.9 Data plane improvement for hierarchical aggregation: Serverful (`SF`), Serverless (`SL`), and LIFL. `SL`'s latency includes contributions of `+SC` (sidecar) and `+MB` (message broker). 169
6.10 Message queuing overheads. 171
6.11 Improvement with LIFL's orchestration, with (i) being additions to baseline LIFL; *x-axis is the number of model updates arriving at the aggregation service concurrently.* 174
6.12 **ResNet-18:** (a) Time-to-accuracy and (b) Cost-to-accuracy; **ResNet-152:** (c) Time-to-accuracy and (d) Cost-to-accuracy. 177
6.13 **ResNet-18 (a, b, c), ResNet-152 (d, e, f):** Time series of arrival rate, number of active aggregators, and cumulative CPU time (seconds) per round. 178

7.1 The architecture of 5GC control and data planes. The 5G UPF performs PDR lookup (based on linked list) to enable data packet forwarding between the RAN and data network. 191
7.2 An architectural overview of $L^2$5GC+'s control plane and SBI. Each control plane NF can support multiple user sessions (each represented as a distinct thread) concurrently. 195
7.3 Synchronous I/O primitives from $L^2$5GC+'s socket APIs: Read() and Write(). Note that the circled number (*e.g.,* (1)) represents the steps of Write() procedure (left half). Otherwise, it represents the steps of Read() procedure (right half). 197
7.4 Concurrent user session support in $L^2$5GC+. Control plane messages to different user sessions are de-multiplexed at the I/O stack after user session table lookup. 204
7.5 (Top) The logical topology, and (Bottom) a snapshot of commercial testbed setup in $L^2$5GC+. RU: Radio Unit; DU: Distributed Unit; CU: Central Unit; CN: Core Network; PoE: Power over Ethernet; 5 laptops as UEs with 5G dongles 209
7.6 Latency from 5G CN, tested with ***commercial*** UEs and RAN. 210
7.7 Total latency (including core network and UE/RAN) of control plane events tested with ***commercial*** UEs and RAN. 211
7.8 Total latency of various control plane events with ***simulated*** UEs and RAN. *x-axis*: number of UEs. 213

7.9 PDR lookup comparison: (a) PDR lookup latency with increasing # of PDR rules; (b) Throughput with increasing # of PDR rules (Packet size 68 bytes) 214

List of Tables

3.1 Qualitative Comparison on the Features of different CNI Plugins. 25
3.2 HTTP Performance of CNIs for 1 & 400 clients 64
3.3 Number of iptables chains and rules with default CNI configurations. Inter-host case collects from both the source host and destination host. 66

4.1 Experiment configuration 85
4.2 Comparing Mu with the standard Knative build 86

5.1 Per request Knative overhead auditing of data pipelines for a '1 broker/front-end + 2 functions' chain. 99
5.2 Per request data pipeline overhead for SPRIGHT 124
5.3 Latency comparison at 5K and 25K concurrency 132

Chapter 1

Introduction

Cloud computing has become increasingly popular, with a growing number of users relying on it for a variety of emerging applications, including 5G networks [176] and machine learning. These applications benefit from the cloud's ability to provide scalable, flexible, and cost-effective computing resources [166, 181]. The rise of 5G, with its need for high-speed data transmission and low-latency communication, and machine learning, which demands significant computational power for training complex models, has further underscored the importance of cloud computing.

In the midst of this growing reliance on cloud infrastructure, a new paradigm known as serverless computing, also known as Function-as-a-Service (FaaS [197, 109]), has emerged [181, 180, 171, 76]. Serverless computing revolutionizes the way applications are deployed and managed in the cloud by abstracting away the underlying infrastructure [92]. This model shifts the responsibility of managing servers, scaling and capacity planning from the user to the cloud provider [166, 136]. As a result, developers can focus on writing

code and developing applications without having to worry about server configuration and maintenance [166, 136].

However, serverless computing faces several significant challenges that impact its effectiveness and scalability. One of the primary issues is the heavyweight data plane, which often relies on kernel-based networking and container-based sidecars [181, 215, 138]. This design can introduce considerable overhead, resulting in higher latencies that are detrimental to real-time and interactive applications such as those in the Internet of Things (IoT) and edge computing [215]. Another challenge is the slow startup times of serverless functions, commonly referred to as "cold starts" [132, 156]. When a function is invoked after being idle, it often requires additional time to initialize, which can lead to noticeable delays. This latency is problematic for applications requiring immediate responsiveness, as it can degrade user experience and hinder performance [132, 156, 180]. Furthermore, the placement of serverless functions are often suboptimal [166, 138]. Current serverless platforms typically do not account for the specific resource constraints and performance requirements of different applications. This can lead to inefficient use of resources and increased operational costs [166, 183]. Additionally, poor function placement can exacerbate latency issues, especially in distributed environments where data and compute resources are geographically dispersed [138].

Addressing these challenges is crucial for advancing serverless computing to better support a wider range of applications, particularly those requiring low-latency and high scalability. In this dissertation, we first start with optimizing the data plane, improving function initialization processes, and developing more intelligent function placement algo-

rithms are essential steps toward overcoming these hurdles and realizing the full potential of serverless computing. This involves the design of (1) a placement engine that considers heterogeneity and fairness among competing functions [166]; (2) a detailed examination of Container Network Interface (CNI) overheads, highlighting the heavyweight nature of the kernel networking stack [178, 179]; (3) a novel, event-driven proxy leveraging the extended Berkeley Packet Filter (eBPF) to replace traditional container-based sidecars [215, 181, 180]; and (4) the design of event-driven shared memory processing for a significant enhancement in the dataplane scalability of serverless function chaining [181, 180]. Through evaluations of our design with various serverless workloads such as online web services and IoT, we show the benefit of a careful design of the serverless computing environment. We achieve an order of magnitude improvement in throughput and latency, while substantially reducing CPU usage, and mitigating the need for 'cold-start'. We show how our design can also be very suitable to support Federated Learning, to achieve considerable savings on training time and cost [183]

In addition, the integration of Network Function Virtualization (NFV) into cellular networks, adopted by operators, vendors, and standards bodies (e.g., 3GPP), has resulted in the fundamental transformation of the 5G cellular core (5GC) architecture [130]. The cellular core, in essence, serves as the central anchor point between user equipment (UE) and the data network (e.g., Internet), managing crucial control and data plane functions, such as user authentication and mobility management, user packet routing and classification. Having the 5GC NFs implemented as cloud-native microservices promises increased flexibility, scalability, and ease of deployment [159].

However, the design of the data plane and control plane in the 5G Core (5GC) network presents several challenges that hinder its optimal performance in cloud environments. The data plane of 5G lacks a scalable and fast packet classification mechanism, which is essential for meeting the low-latency requirements of 5G applications [131, 182, 130]. This deficiency results in suboptimal packet processing speeds, thereby affecting the overall performance of the network. Moreover, the 5G control plane has adopted a cloud-native design through the use of the Service-Based Interface (SBI [143]). This interface facilitates the disaggregation of 5GC control plane network functions (NFs), allowing for greater flexibility and scalability in deployment. However, the current design of the SBI is inefficient, leading to significant processing delays in control plane events. These delays, often overlooked, can have a substantial impact on the performance of the 5G network. The inefficiencies in both the data plane and the control plane design mean that the 5G core network struggles to achieve the best possible performance in cloud settings. Addressing these issues is crucial for improving the performance of 5G in the cloud. In this dissertation, we enhance the data plane with more efficient packet classification mechanisms [182, 130] and optimizing the SBI design to reduce processing delays can help achieve the low-latency, high-performance characteristics that 5G networks require [159]. This will enable 5G to fully leverage the benefits of cloud-native architectures and meet the demanding requirements of modern applications.

This dissertation makes the following major contributions:

Contribution 1: Assessing Container Network Interface. Container Network Interface (CNI [6]) is the fundamental building block of container networking. Given that serverless computing relies heavily on containerization to achieve necessary isolation and

virtualization, dissecting the CNI helped us understand the networking overhead in serverless computing. To this end, we conducted a comprehensive study of various CNI solutions to understand their implementation details and evaluate their performance [178, 179]. The key insight was the heavyweight nature of the kernel networking stack, which causes most of the container networking overhead. The use of advanced host networking techniques, such as extended Berkeley Packet Filtering, can greatly improve the performance of container networks. However, eBPF requires careful design to truly realize its value. This perspective guided our design of the high-performance data plane for serverless computing.

Contribution 2: An Efficient and Fair Algorithm for Serverless Function Placement. Serverless platform, e.g., Knative [59], has the capability to flexibly scale and schedule the cloud services, which does help to improve the resource utilization and relieve the intensive resource concerns in edge cloud. Whereas, resources in the cloud are heterogeneous, saturating only one resource type will lead to the under-utilization and unfairness of other types of resources, which eventually has negative effects on the overall resource utilization. We first understand the unfairness of the function placement logic in current serverless computing frameworks, which are not primarily designed for resource-constrained edge environments, and fail to satisfy the Service Level Objectives (SLOs) for co-locating serverless functions. We then propose Mu [166], which includes a serverless function placement algorithm that considers resource demand across multiple dimensions (CPU, memory). We adapted the notion of dominant resource fairness to ensure efficient and fair placement of functions in resource-constrained edge environments.

Contribution 3: High-performance and Efficient Serverless Computing using Event-driven Shared Memory Processing. We proposed SPRIGHT [181, 180] that focuses on addressing the inefficiencies and performance challenges observed in the data plane in existing serverless frameworks. SPRIGHT's contributions include the design and implementation of event-driven shared memory processing for serverless function chaining, through the extensive use of the extended Berkeley Packet Filter (eBPF [39]). This design minimizes unnecessary protocol processing and serialization-deserialization overhead in the serverless data plane. eBPF's use ensures that shared-memory processing is event-driven compared to polling-based shared memory processing that constantly consumes the CPU cycles [225, 185], thus keeping network overhead strictly load-proportional. We also introduced the eBPF-based event-driven sidecar [215] as an alternative to the heavyweight, container-based sidecar, which causes significant overheads in current serverless frameworks.

Furthermore, the problem of slow startup of serverless computing occurs when functions need to be started on demand and the infrastructure needs time to allocate resources and execute the function [156, 132]. The negative impact of slow startup is especially evident in scenarios that require fast or real-time responses. Addressing the challenge of slow startup in serverless computing is critical to ensure optimal performance and cost-effectiveness in a variety of application scenarios. To this end, we investigate the impact of container networks on the startup of serverless functions, especially in the context of massively paralleling the startup of multiple serverless functions [132, 180]. This study identified the major bottlenecks in container networks that cause delays in function launches, which guides our design philosophy for mitigating function startup latency, which is to keep

functions warm at the lowest possible cost, thus avoiding the need for fast startups while enabling real-time responses.

Contribution 4: Efficient Model Aggregation for Federated Learning using Serverless Computing. Federated Learning (FL) typically involves a large-scale, distributed system with individual user devices/servers training models locally and then aggregating their model updates on a trusted central server [87, 126]. Existing systems for FL often use an always-on server for model aggregation, which can be inefficient in terms of resource utilization [87, 126, 98]. They may also be inelastic in their resource management. This is particularly exacerbated when aggregating model updates at scale in a highly dynamic environment with varying numbers of heterogeneous user devices/servers [87].

We present LIFL [183], a lightweight and elastic serverless cloud platform with fine-grained resource management for efficient FL aggregation at scale. LIFL is enhanced by a streamlined, event-driven serverless design that eliminates the individual heavy-weight message broker and replaces inefficient container-based sidecars with lightweight eBPF-based proxies. We leverage shared memory processing to achieve high-performance communication for hierarchical aggregation, which is commonly adopted to speed up FL aggregation at scale. We further introduce locality-aware placement in LIFL to maximize the benefits of shared memory processing. LIFL precisely scales and carefully reuses the resources for hierarchical aggregation to achieve the highest degree of parallelism while minimizing the aggregation time and resource consumption. Our experimental results show that LIFL achieves significant improvement in resource efficiency and aggregation speed for supporting FL at scale, compared to existing serverful and serverless FL systems.

Contribution 5: High-performance, 3GPP-compliant 5G Core Network. The control plane in 5G core (5GC) presents challenges due to inefficiencies in handling control plane operations (including session establishment, handovers and idle-to-active state-transitions) of 5G User Equipment (UE). The Service-based Interface (SBI) used for communication between 5G control plane functions introduces substantial overheads that impact latency [130]. Typical 5GCs are supported in the cloud on containers, to support the disaggregated Control and User Plane Separation (CUPS) framework of 3GPP. L25GC [130] is a state-of-the-art 5G control plane design utilizing shared memory processing to reduce the control plane latency. However, L25GC [130] has limitations in supporting multiple user sessions and has programming language incompatibilities with 5GC implementations, e.g., free5GC [58], using modern languages such as GoLang [70]. Further, new and constantly evolving use cases continue to place performance demands on the 5G data plane, especially for low latency communications with a large number of concurrent user session. This requires to re-examine the packet classification mechanisms to support fast packet processing in the User Plane Function (UPF) [130]. This work lies on both the control plane and data plane of the 5G core network (5GC), with the aim of reducing network overhead and thus providing low-latency 5G services to end users. This work includes optimization of the data plane and control plane of the 5GC:

- ***Fast and Scalable Packet Classification in 5GC User Plane Functions.*** In the data plane, we tackled the challenge of expensive packet classification in the 5G User Plane Function (UPF) [130]. UPF is an important component of the 5GC data plane, which interconnects the radio access network with the Internet. The slowdown

in packet classification arises from the surge in user sessions, as the conventional linear search method recommended by 3GPP proves to be unscalable. My contribution centered on devising faster approaches for packet classification in the 5GC data plane [130]. This addressed the critical need for maintaining high performance in 5G UPF in the face of an increasing number of user sessions, ensuring efficient packet processing for diverse services and use cases. The advancement in this work is also applicable to 6G and beyond systems to compensate for the shortcomings of the 3GPP-specified cellular network design [182, 131].

- ***High-performance, 3GPP-compliant Service Based Interface.*** We addressed the challenges faced by the 5GC control plane, which relies on the 3GPP-specified Service-Based Interface (SBI) to exchange control events between 5G functions [130]. However, the existing SBI introduced significant overhead that impacted the latency of the 5GC to process control events from end-users. To overcome these challenges, we redesigned the SBI to be strictly 3GPP compliant while maintaining high performance through the use of shared-memory processing [159]. The key was to augment the asynchronous shared-memory I/O with synchronous data exchange primitives, while adding stateful processing to the shared-memory network stack to differentiate between user sessions [159, 184]. The improved design facilitates integration with 3GPP-compliant SBI and scales to multiple users. It optimizes the 5G control plane in terms of latency, scalability, and user experience, as we have demonstrated on commercial 5G testbeds.

The organization of this dissertation is as follows: Chapter 2 discusses related work. Chapter 3 assesses various essential aspects of container networking, including the functionality, performance, and scalability. Chapter 4 presents the fair placement of serverless functions. Chapter 5 presents a high-performance and efficient data plane for serverless computing, using eBPF-based shared memory processing. Chapter 6 describes the efficient model aggregation in Federated Learning using serverless computing. Chapter 7 introduces control and data plane co-optimization in the 5G cellular core. Finally, chapter 9 concludes this dissertation.

Chapter 2

Related Work

2.1 Container Networking

There are a number of blog articles, as well as notes on Github repositories, that provide a good description of different CNIs [103, 101, 210, 17]. Moreover, several works [80, 206, 141, 174, 142, 224] have compared and evaluated the performance of different CNI plugins. Suo et. al. [206] study different container network models and evaluate them across different aspects, such as the TCP/UDP throughput, latency, scalability, virtualization overhead, CPU utilization, and launch time of container networks. While the work attributes the performance differences observed across different CNIs to their different datapaths, it fails to identify the root causes behind these differences, as we have done here. As we observe, the main overhead is from how the CNI plugins interact with the network stack, which heretofore has not been adequately examined.

Kapocius [141] evaluates the performance of Kubernetes CNI plugins on both the virtual machines and bare metal. Kapocius [142] also evaluates several popular CNIs with

different factors considered, e.g. MTU, the number of aggregated network interfaces, and NIC offloading conditions. Their results present the performance variation (at a high level) with different aggregated interfaces and NIC offloading configurations. However, both the papers do not analyze the performance differences, or provide an in-depth analysis of the kernel and namespace overheads observed for different CNIs. Bankston et. al. [80] compare the performance of CNI provided by different public cloud providers (e.g., AWS, Azure, and GCP) with different instances. They also evaluate the impact of encryption and MTU on performance. Their work provides limited insight into the different open-source CNIs. Park et. al. [174] specifically compare the performance of Flannel network, OVS-based network, and native-VLAN network, but again, only at a high level. Ducastel [101] evaluates the most popular CNI plugins using several benchmarks. It also provides a qualitative comparison on security and resource consumption, but is limited to the inter-host case, providing a high-level, throughput-only comparison. Hao et. al. [224] study three container network solutions (Calico, Flannel, and Docker Swarm Overlay) and compare their performance based on TCP/UDP throughput and ping delay. However, the performance difference between different solutions are not explained in detail.

Generally, all these existing works fail to provide a kernel-level analysis and comparison for CNI plugins. We offer an objective comparison across all of the CNIs, from both a qualitative and quantitative perspective, which is important for guiding users as well as the future development of a scalable, high-performance CNI.

2.2 Function Placement in a Serverless Cloud

The bin-packing problem in the context of cloud resource allocation has been investigated with a handful of research efforts. Most of these problems are modeled as different variants of bin-packing problem. [203] addresses the placement of VMs in heterogeneous distributed platforms, in which there are multidimensional physical resources need to be allocated. They formulate the problem through a Mixed Integer Linear Problem (MILP) model and apply different kinds of algorithms, such as Relaxed Solutions, Greedy Algorithms, Vector Packing Algorithms. They also evaluate the impact of inaccurate resource needs estimation and develop an iterative algorithm to mitigate the penalty of error estimations. From their results, the vector packing approaches outperform the other alternative algorithms.

[115] focuses on modeling of energy aware scheduling of VMs, aimed to minimize the power consumption by decrease the number of running VMs and increase the number of idle VMs. They address the problem through a migration algorithm based on Integer Linear Programming model. They take multidimensinal resource constraints into consideration, however, they assume all these constraints are satisfied for simplification.

[220] consider the fluctuations of the resource demands and tend to improve the resource utilization. The authors report the resource usage (CPU usage in their test case) is typically low and rarely becomes high, which indicates the resource demands can be characterized by percentiles of usage history instead of the peak demand. This idea shares the similarity with the resource requests and limits configuration in Kubernetes scheduler. The problem is formulated based on a Mixed Integer Programming model, with the objective

to minimize the amount of allocated resources. The dependability requirements are taken into account in this work by using fault-tolerance mechanisms for server failure and the migration plans during maintenance work.

[106] propose an Ant Colony Optimization (ACO) meta heuristic-based algorithm to optimize the power consumption and resource wastage in VM consolidation. Notably, It is the first work that applies the ACO metaheuristics to address the multidimensional bin-packing problem. The problem is formulated as a Multi-dimensional Vector Packing Problem. Multiple computing resources (e.g., CPU, memory) are taken into account in this model. Besides, the utilization of different resources are balanced through the proposed algorithm, which addresses the concern when dealing with multiple resource types. That is saturating only one resource type would likely lead to other types of resource underutilized, as the improvement on the resources utilization will stop if one of the resource type has been saturated.

In [112], the authors develop a dynamic bin-packing model instead of the static version, which fits better with real data center environment. In the dynamic bin-packing model, the arrival of items have randomness over time. After the items complete their tasks, they will be terminated and leave the system. Through their experimental analysis, they demonstrate that the Best Fit algorithm achieves the best performance when dealing with the bin-packing problem in practice. This work also considers the resources in different dimensions. They treat different resource types with different weights. However, the impact of weights on the performance of is not reported in this paper.

Some works focus on the greedy and random based algorithms to address the multi-dimensional bin-packing problem [205, 204]. Based on their results, the proposed algorithm are proved to be asymptotically optimal. Ghaderi [111] considers to schedule VMs in a multi-server system, which is common in today's cloud data center. A class of randomized algorithms aimed at VM placement are presented. The algorithms are proved to be able to achieve maximum throughput without preemptions. The algorithms are distributed in nature due to the distributed queueing architecture and they can be implemented with low complexity. However, the multidimensional model used in this paper doesn't fit with the various resource types. The "dimension" in this work denotes the various VM types rather than resource types. A weighted strategy is applied to deal with different VM types and help to figure out the best score of placement scheme.

2.3 Optimizing Serverless Computing Data Plane

In recent years, a number of serverless platforms have been launched, *e.g.,* AWS Lambda [47], IBM Cloud Functions [25], Apache OpenWhisk [22], OpenFaaS [30], Knative [59], *etc*, to support cloud-resident applications. Work on understanding the performance impact of commercial or open-source serverless platforms [151, 81] has guided us on the design of SPRIGHT. Li *et al.* [151] showed that the overhead of the ingress gateway reduced the throughput by 13%, compared to the performance of function invocation using the 'direct call' mode (*i.e.,* the client directly invokes the function instance, bypassing the ingress gateway). Priscilla *et al.* [81] studied the suitability of different serverless function startup modes (*i.e.,* cold and warm) for supporting IoT applications, indicating that

cold start can have significant resource-saving benefits but can impact response time. This prompts us to examine the resource consumption of each component carefully.

Several past works have examined the inefficiency and overheads that exist in Linux networking, including data copies and context switching [90, 167, 148, 150]. The overhead of protocol processing [178] and serialization-deserialization [218, 140] directly impact networking performance, which applies to the container-based serverless function, including function chains. A variety of optimizations have been proposed to improve the network performance for different application scenarios, which can be complementary to current Linux networking (*e.g.,* XDP [125], AF_XDP in OVS [211]) or bypass kernel-based networking (*e.g.,* NetVM for NFV [127]). Our work combines the advantages of kernel-bypass zero-copy networking where essential for serverless function chains, and leveraging eBPF-based event-driven processing.

Multiple proposals optimize different aspects of serverless frameworks, *e.g.,* resource provisioning, function deployment, load balancing [200, 166, 207, 86, 173, 139, 138], runtime overhead reduction [74, 76, 197, 110, 171, 76], and mitigation of function startup delay [108, 196, 156, 109, 193, 212, 214] within serverless platforms. Further, [173], [86], [207] aim to optimize resource allocation and deployment of serverless functions on the basis of a chain, which improves the efficiency and flexibility of building microservices using serverless function chaining. However, they do not focus on optimizing the dataplane, which as we show has a significant impact. Furthermore, substantial efforts have been directed towards addressing the data plane overheads inherent in serverless architectures [181, 136, 197, 223], characterized by heavyweight function chaining and sidecar proxy.

2.4 'Cold Start' in Serverless Computing

The cold start latency of serverless functions detracts from their being an ideal framework for building microservices. [108] proposes a startup latency optimization specifically for Kubernetes-based environments by placing pods on nodes that have container image dependencies locally to avoid the latency of pulling images. However, their 95%ile startup latency after optimization is still around 23s, severely impacting the QoS. In addition, startup (either cold start or pre-warm [196]) adds additional costs, as we have observed, making optimizations built around cold start less desirable. A policy of 'keep-warm' of pods has been an alternative to mitigate the cold start latency in serverless [156]. They can achieve an 85% improvement of the 99%ile latency. Although [156] considerably improves the SLOs, it is built on Knative with heavyweight components (*e.g.,* queue proxy), resulting in excessive resource usage. Fuerst *et al.* [109] consider greedy-dual caching to determine which functions should be kept as warm. By factoring in several key indicators of a function, *e.g.,* memory footprint, invocation frequency *etc.*, they can prioritize functions to be kept warm, thus limiting memory consumption to keep a minimum number of warm functions and achieve SLOs. Since SPRIGHT primarily contributes to controlling CPU usage, [109] can be a good complement to SPRIGHT to reduce memory utilization.

2.5 Federated Learning Optimization

As a fast-evolving ML technology, a large body of work has been proposed for FL; the proposals in [162, 152, 169, 153, 189] focus on FL algorithms while others investigate how to select FL clients or datasets more intelligently [146, 157, 72, 137, 170, 198, 118, 147, 104].

[158] seeks to schedule FL jobs across a shared set of FL clients with less contention and reduce job scheduling delays. These efforts are orthogonal to our effort (LIFL) because LIFL focuses on system-level optimization of model aggregation of FL. This makes LIFL a good complement to these efforts by providing an efficient and high-performance FL system to bring various FL approaches to the ground.

Several open-source FL platforms, *e.g.,* Flame [57], FATE [40], OpenFL [43], FedML [123], IBM federated learning [160] have been launched to facilitate the promotion and adoption of FL in both research and applications. These platforms assume themselves to be a serverful design with static, inflexible deployment, which makes them unprepared for large-scale FL. Our design of LIFL can be used as a representative case to guide the future development of these platforms.

2.6 Optimizing 5G Core Control Plane

There has been a focus on how to reduce the latency of 3GPP SBI in the 5GC control plane. $L^2$5GC [130] is the state-of-the-art 5GC control plane optimization that seeks to use shared memory processing to reduce the control plane messaging latency incurred by kernel-based 3GPP SBI, which is commonly adopted in existing 5GC implementation, such as our earlier work free5GC [58]. Although $L^2$5GC achieves considerable latency reduction of various control plane events, its imperfect design of shared memory I/O, *e.g.,* lack of synchronous data exchange support, unawareness of connections, making it ill-suited for a 3GPP-compliant 5GC control plane and fail to scale up to multiple user sessions. Buyakar *et al.* [143] propose to replace the HTTP/REST APIs with gRPC to construct 3GPP SBI,

since gRPC shows better scalability, in terms of CPU utilization and data transmission latency, compared to the HTTP/REST APIs when dealing with an increasing number of UEs. However, gRPC still suffers from kernel networking overhead as HTTP/REST-based SBI, making it less competent compared to our improved 5GC, $L^2$5GC+.

Apart from optimization on 3GPP SBI, there are many other efforts on optimizing the cellular core control plane. Neutrino [75] is a 5GC control plane design that also seeks to reduce control plane latency. However, unlike $L^2$5GC+ that focuses on reducing the messaging latency within the 5GC control plane, Neutrino attempts to reduce the messaging latency between the RAN and the 5GC control plane by minimizing the data serialization overhead, while being 3GPP-compliant. This could be a good complimentary to $L^2$5GC+.

CleanG [168] and DPCM [154] reduce the latency of 5GC control plane by re-designing control plane procedures. CleanG primarily focuses on creating a new control plane protocol that can simplify the control plane interactions in cellular networks, thus reducing latency [168]. Another approach, DPCM [154], reuses the UE-side state to skip unnecessary control plane procedures (*i.e.,* those used to generate the UE-side state which is already there). However, both of these proposals are not 3GPP-compliant, which makes them less complimentary to $L^2$5GC+, as our faith of $L^2$5GC+ is to keep 3GPP compliance.

Besides latency optimization, [163] seeks to characterize and model the control plane traffic in cellular cores, which may facilitate the testing and evaluation of $L^2$5GC+'s control plane design when real traffic is not available due to regulatory compliance.

Chapter 3

Assessing Container Network Interface: Functionality, Performance, and Scalability

3.1 Introduction

Kubernetes is the leading container orchestration platform used by cloud service providers (CSPs) to improve the utilization of their cloud resources [82]. It provides the flexibility to run a variety of containerized cloud applications, with the ability to deploy on both physical and virtual cloud resources. In Kubernetes, a “Pod” is the atomic unit of deployment, for scaling and management [15]. A pod may comprise one or more containers that share the same resources including the networking context. Pods can be scaled to multiple instances to meet the workload characteristics and also to provide failure resiliency.

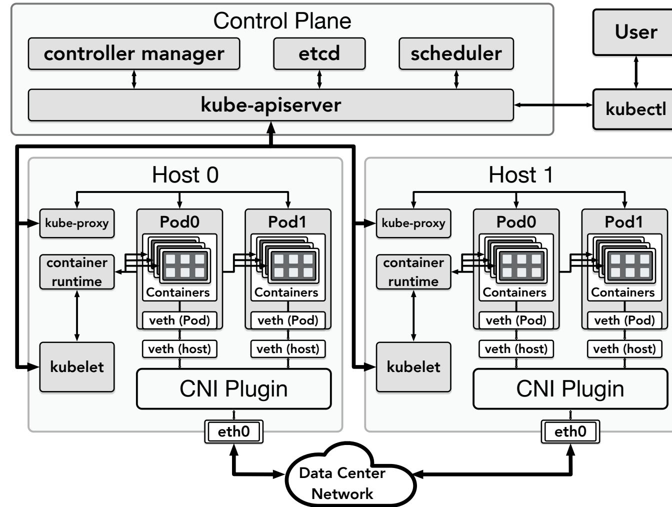


Figure 3.1: Kubernetes Cluster and the Role of CNI.

The proliferation of microservices [213] and function-as-a-service [92] architectures for deploying cloud-based services make it necessary to support large numbers of containers, and to provide efficient communication between them. Hence, orchestration and networking are critical and need to be automated, scalable, and secure to benefit large production deployments. Kubernetes adopts the Container Network Interface (CNI) specification as its core networking foundation [6]. Each Pod in a Kubernetes cluster is given a unique IP address for communication.

A general architecture overview of a Kubernetes cluster is shown in Fig. 3.1, with one control plane host and two worker hosts. The control plane host is in charge of maintaining the cluster state, and the worker host is responsible for running cloud workloads in the execution unit named Pod, while the CNI Plugin facilitates communication among these execution units [18]. There are a number of different CNI implementations, and these CNI ‘plugins’ perform the tasks for Pod networking in a Kubernetes cluster. With thousands

of Pods running in a cluster, the network's status can change rapidly, with frequent creation and/or termination of pods. When a new Pod is added, the CNI plugin coordinates with the container runtime and connects the container network namespace with the host network namespace (e.g., by setting up the virtual ethernet (veth) pair), assigns a unique IP address to the new Pod, applies the desired network policies and distributes routing information to the rest of the cluster.

Several open-source CNI plugins are available for use in a Kubernetes environment. Amongst them, Flannel[107], Weave Net (or Weave)[217], Cilium[96], Calico[68], and Kube-router[144] are popular and have been adopted by many Kubernetes distributions [101]. While each finds their application in different contexts due to their unique and distinct networking characteristics, we believe there is an inadequate understanding and a lack of a comprehensive characterization of these different CNI plugins on both the qualitative and performance aspects. To better understand the operations of different CNI plugins, it is necessary to generalize the working of the different CNI plugins based on the underlying network model and identify the key implementation differences and their corresponding impact on the performance and scalability aspects.

While existing works [206, 141, 80, 174, 142, 224] study the overall performance of different CNI plugins at a preliminary level, there is still a lack of in-depth understanding on how the various design considerations affect performance. Besides, there is also a lack of examination on the CNI plugins' performance under large scale deployments. With the help of autoscaling feature, the number of Pods on a single host can be easily scaled up to hundreds [2], which results in additional interference and impact on the performance of the

CNI plugins. Hence an analysis of the impact of background communication for different CNIs is necessary to understand scalability, and associated performance impact.

In this chapter, we provide an insight into the overall performance of Pod networking with different CNI plugins, by examining throughput, latency, and fine-grained CPU measurements of the various components of the networking stack. First, we present a qualitative analysis of the popular CNI plugins, namely: Flannel, Weave, Cilium, Calico, and Kube-router, to provide a high-level operational view of the feature support of different CNIs. We also consider the different variants of the Calico to demonstrate the effect of tunneling and overheads with different encapsulation modes. We omit the other less frequently used (and some outdated) CNIs such as Romana [14], Canal [3], and Contiv-vpp [8] from our study. Next, we provide a measurement-driven quantitative analysis of their performance for various communication modes. To summarize, the contributions of our work include:

- We provide qualitative analysis for different CNI plugins in terms of the subset of network or datalink layer features they support (*e.g.,* IPv6, encryption support), and accordingly classify and generalize the CNI Plugin networking model into four different classes.
- We analyze the interactions with the host networking stack including the network filter configurations (iptables rules) across different dimensions (*e.g.,* iptables chains, packet forwarding, overlay tunneling, extended Berkeley Packet Filter (eBPF), *etc.*) to determine the critical function calls that contribute to overhead.
- Based on this qualitative analysis, we examine the root cause for the performance differences across several representative CNI plugins for packet transmission throughout

the entire network protocol stack with a measurement-based quantitative evaluation. We analyze the performance impact of various data path components in the kernel network protocol stack.

- We present extensive performance analysis by considering different real-world traffic patterns for varying scale of deployment Pods and concurrent clients to help understand the suitability of using different CNIs. We briefly examine Pod startup latency.

3.2 Background

In Kubernetes, networking plays a pivotal role in enabling the cluster-wide communication among the pods. In this section, we briefly present the details of the Kubernetes networking and CNI plugin models.

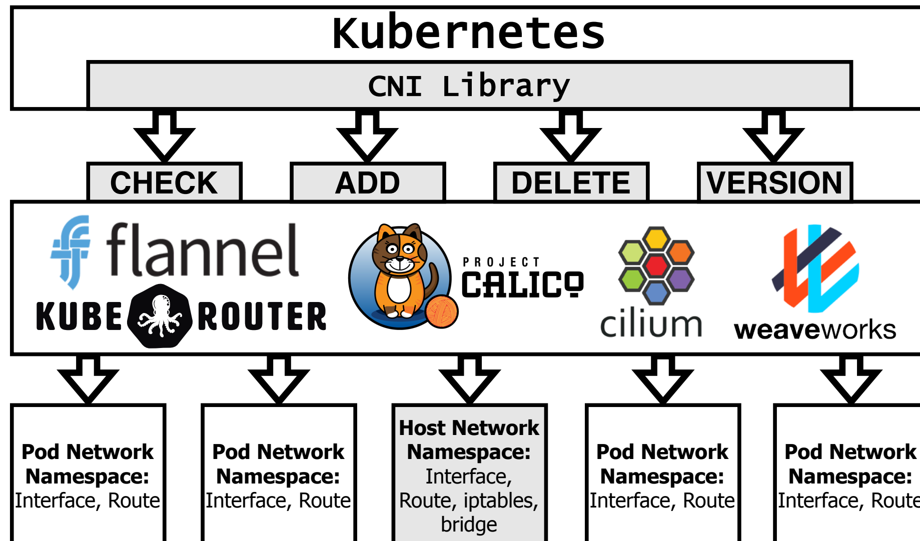


Figure 3.2: CNI Plugin: Interfacing w/Kubernetes & Namespaces

3.2.1 Container Network Interface

The Container Network Interface (CNI) is a container networking specification proposed by CoreOS [6] and has been adopted by several open-source projects such as Cloud Foundry, Kubernetes, Mesos, *etc.* and has also been accepted by the Cloud Native Computing Foundation (CNCF) [4] as an industry standard for container networking[1]. Container Network Model (CNM) is an alternative container networking standard proposed by Docker [7]. Although, both CNI and CNM are modular and provide plugin-based interfaces for network drivers to create, configure, and manage networks, the CNM is designed to support only the Docker runtime, while CNI can be supported with any container runtime [91].

Table 3.1: Qualitative Comparison on the Features of different CNI Plugins.

CNI Solutions	***Network Model***	***Tunneling Options***	***Network Policies***		***Packet Encryption***	***Additional Features***	***Default MTU (Bytes)***
			Kubernetes / 3rd party	***Ingress / Egress***			
Flannel	Hybrid + Underlay/Overlay	VXLAN	No / No	No / No	No	–	1450
Weave Net	Hybrid + Overlay	VXLAN	Yes / No	Yes / Yes	Yes	Multicast	1376
Cilium	L3 + Underlay/Overlay	VXLAN/Geneve	Yes / Yes	Yes / Yes	Yes	IPv6	1500
Calico	L3 + Underlay/Overlay	IP-in-IP/VXLAN	Yes / Yes	Yes / Yes	No	IPv6	1440
Kube-router	Hybrid + Underlay/Overlay	IP-in-IP	Yes / No	Yes / No	No	IPVS/LVS, DSR	1500

The CNI specification defines a simple set of interfaces (*e.g.,* CHECK, ADD, DELETE) for adding and removing a container from the network. With the help of CNI plugin, network interface, route, iptables, *etc* can be efficiently set up in a Pod/host network namespace. A modular/driver-based approach allows the integration with several 3rd party implementations of the CNI specification, called the CNI plugins. A CNI plugin is implemented as an executable and the container runtime is responsible for invoking this

[1]CNCF is backed by a large number of companies that currently support CNI being the de facto standard for container networking [192].

plugin to set up and destroy the container network stack as shown in Fig. 3.2. The CNI plugin is then responsible for IP Address Management (IPAM), to connect the Pod network namespace with the host network, provide IP address allocation to the container network interface, manage the IP address allocation across different pods in the cluster, configure the routes on both the host and Pod network namespaces, *etc.* "CNI plugins" comprise of two major components, namely the CNI daemon and CNI binary files. The CNI daemon is mainly used to do network management jobs, such as updating the routing information of the hosts[2], maintaining network policies[3], renewing the subnet leasing[4], Border Gateway Protocol (BGP) updating[5], and maintaining some self-defined resources (*e.g.,* IP-Pool in Calico[6]). The CNI binary files are mainly used to create the network devices (*e.g.,* Linux bridge) and allocate IP address to Pods. Although it is possible to concurrently support multiple CNIs for a Pod (*i.e.,* setup a Pod with multiple network interfaces, which are managed by different CNIs), we focus on cases having only a single CNI.

3.2.2 Linux Network Namespace and Kubernetes Namespace

Before we delve into the Kubernetes networking model, it is necessary to understand the namespace model and distinguish between the Linux network namespace and Kubernetes namespace and the associated impact on setting up of the CNI.

Linux network namespace is designed for network isolation. Each Pod has its own network namespace, which is isolated from the host network namespace and the network

[2]https://docs.projectcalico.org/reference/architecture/data-path
[3]https://kubernetes.io/docs/concepts/services-networking/network-policies
[4]https://github.com/coreos/flannel/blob/master/Documentation/reservations.md
[5]https://docs.projectcalico.org/networking/bgp
[6]https://docs.projectcalico.org/reference/resources/ippool

namespace of other Pods. This enables the Pod to operate its own network stack and interfaces without interference and collision with other Pods. When a new Pod is created, the container runtime interface (CRI) creates the network namespace for the new Pod. Thereafter it invokes the CNI plugin, which allocates the IP address for the pod, attaches the virtual Ethernet pair (veth-pair) to link up the Pod's network namespace to the host network namespace, and add the corresponding routing and network policy rules.

On the other hand, the Kubernetes namespace is used to divide the physical cluster into multiple virtual clusters. This enables sharing the physical cluster resources among different groups of users and makes the management of the cluster more flexible. This also means that the Pods in different Kubernetes namespaces are not strictly isolated. Kubernetes starts with several system namespaces typically prefixed with 'kube-' (*e.g.,* kube-system, kube-pubilc, kube-node-lease, and default). Users are also allowed to create their own Kubernetes namespaces to isolate their workloads from other users. When creating the user namespaces, users can specify the resource quota for the created namespace, such as the maximum number of running Pods, CPU, and memory limits to avoid the threat of exorbitant resources requests. The Kubernetes namespace can be used as a selector in the network policy, to apply a specific network policy to a group of Pods in that namespace, which decouples the Pods from their static IP addresses and improves the resource management efficiency in a large scale cluster.

3.2.3 Kubernetes Networking Model

The Kubernetes networking model is proposed for dealing with four different kinds of communication: i) intra-pod or Container-to-Container communication within a

Pod, ii) inter-Pod or Pod-to-Pod communication, iii) Service-to-Pod communication and iv) External-to-Service communication [5]. To achieve these four communication services, Kubernetes only provides the specification of the network model, while the actual implementation is handed over to the CNI plugins. The key requirements of the Kubernetes network model include i) Pods are IP addressable and must be able to communicate with all other Pods (on the same or different host) without the need for network address translation (NAT), and ii) all the agents on a host (*e.g.,* Kubelet) are able to communicate with all the Pods on that host. CNI plugins may differ in their architecture but meet the above network rules. Thus, there is a range of CNI plugins that adopt different approaches. Popular CNI plugins are Flannel, Weave, Calico, Cilium, and Kube-router [5].

3.2.4 Kubernetes Network Policy

Kubernetes Network Policy is the means to enforce rules indicating which network traffic is allowed and which Pods can communicate with each other. The policies applied to Pod network traffic can be based on their applicability to ingress traffic (entering the Pod) and egress traffic (outgoing traffic). The control strategies include ”allow” and ”deny”. By default, a Pod is in a non-isolated state. Once a network policy is applied to a Pod, all traffic that is not explicitly allowed will be rejected by the network policy. However, other Pods that do not have network policies applied to them are not affected. CNI plugins in Kubernetes can implement elaborate traffic control and isolation mechanisms.

3.3 Qualitative Comparison on CNI Plugins

The different open source CNIs vary in their design and approach towards facilitating intra-host and inter-host Pod communication, and their support for Pod network policies. We provide a careful qualitative analysis of different CNIs based on the layer of operation, packet forwarding and routing approach for Pods within the same host or across hosts. CNI performance and scalability are influenced by the overheads in the network protocol stack (typically in the kernel). We also consider several other factors such as support for encryption, IPv6, and multicast functionalities.

Network model: As Pods are uniquely identified by their IP addresses, CNIs primarily operate at Layer-3 (Network) to facilitate inter-pod communication. However, CNIs can also operate at Layer-2 (Link) for intra-host Pod communication *e.g.,* using the 'Bridge' or 'MacVLAN' capabilities. Layer-2 CNIs take advantage of a software Linux bridge. This greatly simplifies the configuration and management of Pod networking, especially for the intra-host Pod-to-Pod communication. However, they suffer from scale limitations and also exhibit significant time to adapt to the network changes because of MAC learning and forwarding updates. Layer-3 CNIs, based on IP routing and forwarding, are better for scalability and to support cluster-wide communication. They often depend on Border Gateway Protocol (BGP) support to cross autonomous system (AS) boundaries, which could be a potential security concern in some cloud sites [89]. BGP suffers from its security vulnerabilities, such as BGP hijack [195], route leaks [202], *etc.* Alternatively, the 'hybrid Layer-2/Layer-3 solutions' (*i.e.,* Layer-2 for intra-host and Layer-3 for inter-host) may facilitate efficient intra-host, and scalable inter-host, communication.

Packet Forwarding and Routing: Another aspect to consider is the process for packet forwarding between hosts, which has a significant impact on the performance (*i.e.,* latency and throughput). Based on how the packets are forwarded across hosts, the CNI can be classified into two categories: (i) Overlay networking is a virtual network that is built on top of an underlying physical infrastructure, which not only provides new isolation or security benefits but also gets around the dependency (e.g., IP address, routing, etc) on the underlying infrastructure support. All the hosts in an overlay network communicate with each other via the virtual links, which are also called overlay tunnels. When packets go through the overlay tunnel, they are encapsulated with an outer header based on the adopted overlay protocol, such as Virtual Extensible LAN (VxLAN), Generic Routing Encapsulation (GRE), *etc.* Although using overlay technologies remove the dependency on the underlying infrastructure, it increases the difficulty to trace data packets when errors occur in addition to the performance reduction. (ii) Underlay networking is just the datalink layer or Layer-3 (IP) infrastructure for packet forwarding. The underlay network provides the native routing connectivity between different hosts in the cluster. This typically requires BGP, where the host hosts act as BGP peers and share the routing information among them to support inter-host routing without the need for any encapsulation.

Based on the existing implementations of different CNI Plugins, we classify the networking models and datapath design into the following four broad classes: i) Layer-3 + Overlay; ii) Layer-3 + Underlay; iii) Hybrid + Overlay; iv) Hybrid + Underlay.

Layer-3 + Overlay uses an overlay tunnel endpoint (OTEP) and multiple veth-pairs (Fig. 3.3). The OTEP is used to encapsulate/decapsulate the packets. The veth-pair enables data

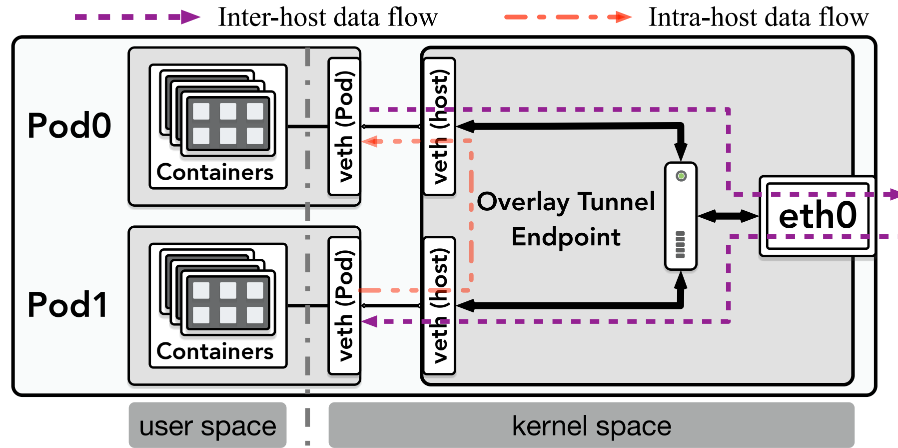


Figure 3.3: Network model for CNIs operating at Layer-3 in overlay mode (*e.g.,* Calico, Cilium)

exchange between the Pod network namespace and the host network namespace. The intra-host data exchange is handled by the host protocol stack in Layer-3. For inter-host case, the outgoing data packet is delivered to the OTEP via IP forwarding, where the packets get encapsulated at the endpoint and sent to its destination via the host's physical interface.

Layer-3 + Underlay comprises of veth-pairs (Fig. 3.4). The intra-host data exchange with IP routing works the same as in the overlay-based design. The inter-host communication is also processed in the host at Layer-3.

Hybrid + Overlay combines the hybrid Layer-2/Layer-3 design with overlays. It consists of a Linux bridge, an OTEP, and multiple veth-pairs (Fig. 3.5). A Linux bridge in the host network namespace connects with the Pods through veth-pairs facilitating intra-host data exchange. For inter-host data exchanges, the outgoing data packet first arrives at the bridge and is then handed over to the host protocol stack operating at Layer-3. The host

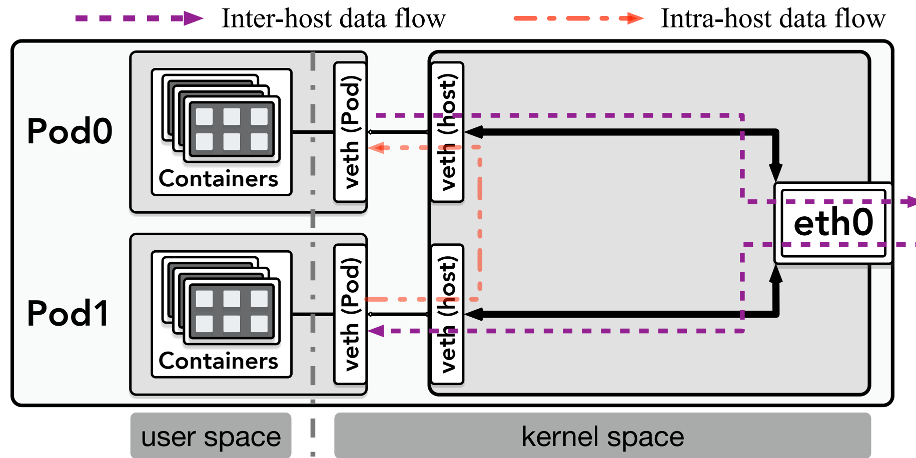


Figure 3.4: Network model for CNIs operating in Layer-3 in underlay mode (*e.g.,* Calico, Cilium)

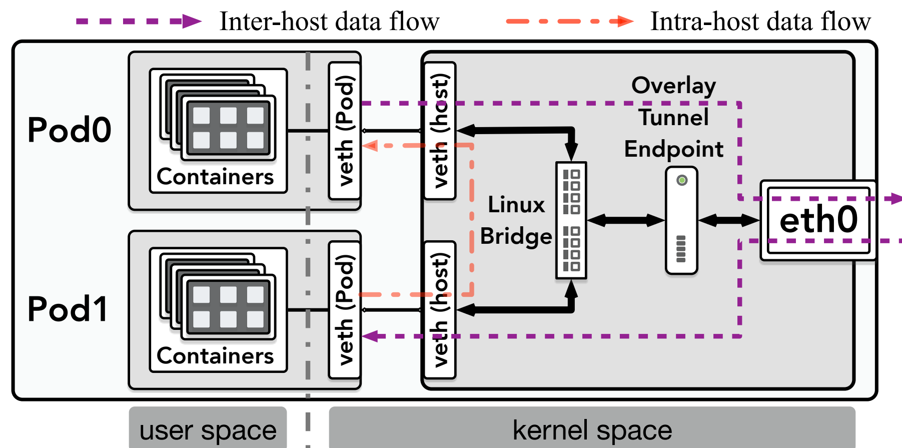


Figure 3.5: Network model for CNIs operating in both Layer-2 & Layer-3 in overlay mode (*e.g.,* Flannel, Weave, Kube-router)

protocol stack forwards the packet to OTEP via IP forwarding. At the OTEP, the packet is encapsulated (based on the overlay encapsulation type) and sent to the destination via host eth0.

Hybrid + Underlay comprises multiple veth-pairs and a Linux bridge (Fig. 3.6). The intra-host data exchange here works the same as in the Hybrid+Overlay approach. For the inter-host communication, the outgoing data packet first arrives at the bridge, which is then handed over to the host protocol stack operated in Layer-3. With the host's IP forwarding turned on, the data packets will be sent through the host's physical interface to the other hosts.

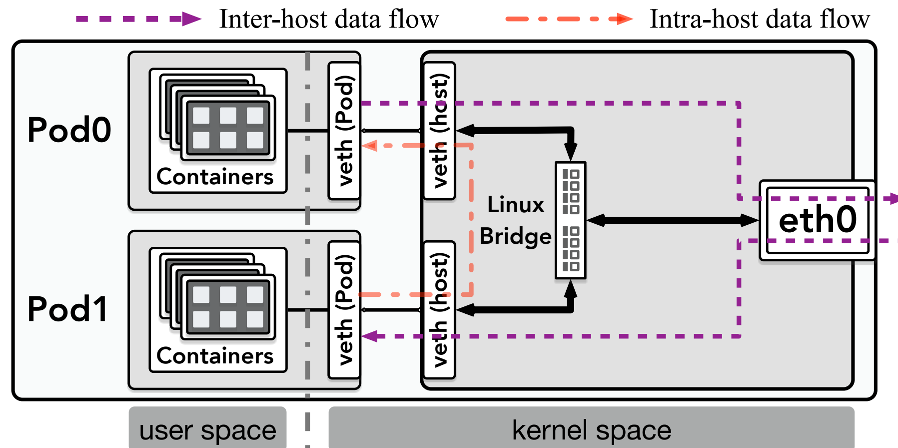


Figure 3.6: Network model for CNIs operating in both Layer-2 & Layer-3 in underlay mode (*e.g.,* Flannel, Kube-router)

Kubernetes Network Policy: Before implementing network policies in a Kubernetes cluster, a network policy controller needs to be installed. This is provided by the CNI plugin. The CNI plugins which support the network policy include Weave, Calico, Cilium, and Kube-router. The CNI plugin daemon works as a Network Policy Controller (a

daemon ensures that some or all hosts in a Kubernetes cluster run a copy of a Pod). Users can provide the Network Policy annotations to the daemon to enable various filtering rules.

3.3.1 Description of each CNI plugin

(1) Flannel:

- **Layer of Operation:** Flannel uses a combination of Layer-2 and Layer-3 operation. Pods in the same host can use the Linux bridge to communicate (Layer-2), while pods on different hosts use an overlay tunnel endpoint to encapsulate their traffic and use Layer-3 routing and forwarding.

- **Packet Forwarding across Hosts:** With its default settings, Flannel configures the Kubernetes cluster with an overlay network. Flannel has several different types of overlay backends (e.g., VxLAN, UDP) that can be used for encapsulation and routing. The default and recommended method is to use VxLAN because of its better performance. The VxLAN backend is in kernel space while the UDP backend works in the user space, which has more context switches (up to 3) compared to the VxLAN mode. Besides the overlay mode, Flannel also has an optional underlay mode, which transfers packets using IP routing (Layer-3). However, this mode requires direct Layer-2 connectivity between communicating containers from the underlying network infrastructure. In both the VxLAN overlay and underlay modes, only one context switch (userspace to kernel space) happens when the packets are delivered from the Pods into the host network stack. However, when using the UDP overlay mode, the packets need to be encapsulated with a UDP header via the Flannel daemon in the

user space. After the packets are delivered from the Pod into the host network stack, it will be delivered back to the userspace to be processed by the Flannel daemon. Then they will enter into the host network stack again to be sent to its destination. Hence, in total three context switches happen under the UDP mode resulting in more CPU overhead and poor performance.

- **Network Policy Support:** Flannel does not implement the network policy controller of its own hence lacks the support to realize any network policies.

- **Miscellaneous:** In the overlay network, each host has its own subnet (a fixed Classless Inter-Domain Routing), which is used to allocate IP addresses internally. When spinning up a new Pod, the Flannel daemon on each host will assign an address to each new Pod from that address pool.

Note: We configure Flannel in VxLAN mode.

(2) Weave:

- **Layer of Operation:** Likewise to Flannel, Weave also operates at both Layer-2 and Layer-3. Intra-host communication is performed at Layer-2, where the packets are forwarded to its destination via the Linux bridge and Layer-3 for the inter-host communication.

- **Packet Forwarding across Hosts:** Weave only uses overlay links (VxLAN or UDP) for communication between hosts. The VxLAN mode is running based on the kernel's native Open vSwitch datapath module, while the UDP mode relies on the Weave CNI daemon to implement encapsulation. Likewise to Flannel, Weave incurs one context

switch when running in VxLAN mode, and three context switches when running in UDP encapsulation mode.

- **Network Policy Support:** Weave provides network policy support for Kubernetes clusters. When setting up Weave, a network policy controller is automatically installed and configured. Weave can only configure the standard Kubernetes Network Policy, which implements network policies on Layer-3 (Network) and Layer-4 (Transport) attributes. Weave implements its network policy based on iptables. Weave uses state extension in iptables, which is a subset of the connection tracking extension (conntrack). Weave uses state extension to speed up the processing of iptables. For an established connection, only the first packet needs to be matched with the iptables, the remaining packets will be allowed to pass directly. Moreover, Weave uses 'ipset' to speedup iptables processing. Ipset uses a hash table to map a rule to a set of IP addresses, and a hash table lookup for a packet to find the target rule.

- **Miscellaneous:** A feature unique to Weave, which is not available in most of the existing CNI plugins, is simple encryption of the traffic based on NaCl (a networking and cryptography library)[7], with a user API to simplify the implementation of encryption in the networking system[8]. Multicast support is provided in Weave to improve throughput and save bandwidth for applications such as streaming video or the exchange of lots of data across multiple containers[9].

Note: We configure Weave in VxLAN mode.

[7] http://nacl.cr.yp.to/
[8] https://www.weave.works/docs/net/latest/concepts/encryption/
[9] https://www.weave.works/use-cases/multicast-networking/

(3) Calico:

- **Layer of Operation:** Unlike Flannel and Weave, Calico operates at Layer-3 for both intra-/inter-host communication.

- **Packet Forwarding across Hosts:** Calico allows for both underlay/overlay packet forwarding across hosts. The underlay mode uses native routing based on BGP. Calico uses BGP to distribute and update routing information providing better scalability and performance[10]. Calico's overlay mode uses IP-in-IP or VxLAN encapsulation. Calico only incurs one context switch for both the overlay modes using IP-in-IP or VxLAN and underlay mode using BGP.

- **Network Policy Support:** Calico has very good features for Kubernetes network policy customization. Users can enable both the Kubernetes network policy as well as Calico's own network policy, covering the policy from Layers 3 to 7. Calico can also be integrated with the service mesh, *Istio*, to implement strategies for workloads within the cluster at the service mesh layer[11]. This means that users can configure iptables rules that describe how Pods should send and receive traffic, thus enhancing the security of the network environment. Calico implements its network policy based on iptables. It inserts user-defined chains on top of the system default chain. Calico uses conntrack to optimize its iptables chains just like Weave. In addition, Calico also uses ipset to provide better scale and performance than default iptables[12].

[10] https://www.projectcalico.org/why-bgp/
[11] https://www.projectcalico.org/category/istio/
[12] https://docs.projectcalico.org/about/about-network-policy

- **Miscellaneous:** Calico provides IPv6 support. However, it is limited to underlay mode only and the overlay (with IP in IP and VxLAN) can support only IPv4 addresses. With IPv6 enabled, the larger address space allows for scalability in the number of endpoints in the Kubernetes cluster.

Note: Calico can be run in two intra-host modes: Calico-wp (native routing with network policy) and Calico-np (native routing without network policy). In 'Calico-wp', we consider the ordering/priority, allow/deny rules, *etc* rules to be enabled. For the inter-host scenario, Calico has two network models ('Layer-3 + overlay' & 'Layer-3 + underlay'). This in conjunction with network policy support provides four different modes namely Calico-wp-ipip, Calico-wp-xsub, Calico-np-ipip, and Calico-np-xsub. "ipip" means the IP-in-IP overlay mode and "xsub" means IP based underlay. Accordingly, we evaluate each of these distinct configuration modes.

(4) Cilium:

- **Layer of Operation:** Cilium is also a Layer 3-only solution. For intra-host communication, Cilium relies on eBPF programs attached at the veth-pairs to redirect packets to their destination. Cilium builds its datapath based on a set of eBPF hooks that run eBPF programs. The eBPF hooks used in Cilium include XDP (eXpress Data Path), Traffic Control ingress/egress (TC), Socket operations, and Socket send/recv. TCs, attached to the veth, are utilized to forward packets through eBPF function calls, *e.g., bpf_lxc*, *bpf_netdev*[13]. Arriving packets at the veth cause eBPF programs to be executed and route traffic.

[13]https://docs.cilium.io/en/v1.7/architecture/

- **Packet Forwarding across Hosts:** Cilium allows for both an underlay (IP) packet forwarding and an overlay solution with VxLAN or Geneve as the encapsulation options. Like Calico, Cilium's underlay mode uses native routing based on BGP. If Cilium uses the overlay solution, *bpf_overlay* will be executed to direct the packet from veth to the OTEP. Like Calico, Cilium also incurs one context switch for both overlay and underlay mode of operation.

- **Network Policy Support:** Cilium supports both the standard Kubernetes network policy and its own network policy customization based on the iptables. These network policies can work in Layers 3-7. In addition, Cilium can use eBPF hooks (e.g., XDP, TC) to define packet filters. Since this filtering occurs earlier than the network protocol stack, it can achieve better performance than iptables.

- **Miscellaneous:** Cilium supports packet encryption in Layer-3, *i.e.,* it provides encryption using the IPSec tunnels. Cilium also provides support for IPv6.

Note: We configure Cilium in VxLAN mode.

(5) Kube-router:

- **Layer of Operation:** Kube-router operates in both Layer-2 and Layer-3. Kube-router uses Linux bridge to forward packets intra-hosts (Layer-2). It uses Layer-3 operation to forward packets inter-hosts.
- **Packet Forwarding across Hosts:** Kube-router allows for both an underlay (IP) packet forwarding across the hosts based on BGP and an overlay solution using IP-in-IP encapsulation. As with Calico and Cilium, packet forwarding in Kube-router incurs just one context switch for both overlay and underlay modes.

- **Network Policy Support:** Kube-router only supports the standard Kubernetes Network Policy APIs to apply at Layer-3 and Layer-4. Kube-router implements its network policy based on iptables. Kube-router uses conntrack to speed up the processing of iptables. Kube-router also uses ipset to get around the overhead of a large iptables.

- **Miscellaneous:** An important feature of Kube-router is the usage of IPVS/LVS kernel features to improve service load balancing performance[14]. Kube-router applies a Direct Server Return (DSR) feature to implement a high-efficient ingress for load balancing, which is a unique feature compared to other CNIs[15].

Note: We configure Kube-router in IP-based underlay mode.

3.3.2 Iptables Comparison

In order to implement specific packet forwarding, routing, and networking policies, the CNI Plugins leverage 'iptables' - a userspace interface to setup, maintain and inspect the tables of IP packet filter (Netfilter) rules in the Linux kernel (we use the terms iptables and Netfilter interchangeably in this paper). Netfilter registers five hook points (callback function points) into the network stack to implement packet filtering, NAT, and security-related policies. The five hook points are: *PREROUTING*, *INPUT*, *OUTPUT*, *FORWARD* and *POSTROUTING*. These are 'chains' of tables of iptable-related processing of a packet, for the purpose of filtering and execution of security policies. Fig. 3.7 shows the iptable chain processing. Each iptables chain comprises a number of tables, each with a different

[14] https://cloudnativelabs.github.io/post/2017-05-10-kube-network-service-proxy/
[15] https://www.kube-router.io/docs/user-guide/

purpose. There are four kinds of tables: *raw*, *mangle*, *nat*, and *filter*. The *raw* table is used to split the traffic without a need for the connection to be tracked. The *mangle* table is used to change the QoS settings of packets. The *nat* table is for network address translation and the *filter* table is used for packet filtering.

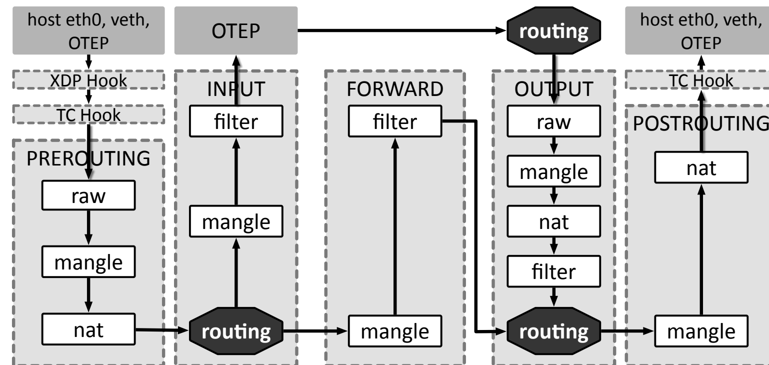


Figure 3.7: Iptables chain processing and hook points [84].

Based on the user-defined network policies for different Pods (input as a *Yet Another Markup Language* file), the CNI's network policy controller configures the iptables on all the Kubernetes worker hosts. When packets arrive at an iptables chain, a *nf_hook_slow()* function call is executed to traverse the list of tables in that chain and process the matching iptable rules. This rule processing can be a major source of overhead in iptables [84].

Moreover, when a host starts up, the kube-proxy installs a set of default iptables rules on the worker hosts. These default iptables rules contribute to the Netfilter processing overhead for packet transmission. These rules are used to redirect the traffic from the default iptables chains to the Kubernetes' user-defined iptables chains (*e.g.,* kube-services, kube-forward, etc). Kubernetes relies on these user-defined chains along with the internal rules to support different kinds of communication. For example, the kube-proxy installs NAT rules to support 'External-to-Service' communication.

Iptables' Routing Management: Based on the network model, different paths will be exercised through the iptable chains. For intra-host communication (Layer-3 IP routing and Layer-2 bridge forwarding), we have the packet processed through the *PREROUTING*, *FORWARD*, and *POSTROUTING* chain. When a source Pod sends out a packet, the packet will be checked first by the *PREROUTING* chain. It will look up three tables (*raw*, *mangle*, *nat*) in the *PREROUTING* chain and match with the rules in these tables. So, if there are any matched rules, the corresponding actions specified will be taken on the packet (e.g., NAT, change ToS (Type of Service) field, etc). The packet will then be checked by *FORWARD* and *POSTROUTING* chain before it arrives at the veth interface of destination Pod. The intra-host routing works for both bridge forwarding and IP forwarding network models, following the same processing. The exception is for the eBPF approach, as the eBPF integrates the packet filtering functions in the eBPF program and bypasses the iptables processing by leveraging the XDP or TC hooks.

In the case of inter-host communication, the traffic can be classified into two different cases: ingress traffic (*i.e.,* inbound traffic originating from an external network and destined towards the internal host network) and egress traffic (outbound traffic originating in the internal network). For ingress traffic in overlay solution, first the packets follow the $PREROUTING \rightarrow INPUT$ path, to be routed from the host Ethernet interface to the OTEP. Then, after the packet is decapsulated by the OTEP, the packets follow the $PREROUTING \rightarrow FORWARD \rightarrow POSTROUTING$ path, to be routed from the OTEP to the destination Pod. For egress traffic in overlay solution, the packets follow the $PREROUTING \rightarrow FORWARD \rightarrow POSTROUTING$ chain first, when routed from

the Source Pod to the OTEP. Next, the packets follow the $OUTPUT \rightarrow POSTOUTING$ route, to be routed from the OTEP to the host Ethernet interface. The configuration for the underlay solution has the $PREROUTING \rightarrow FORWARD \rightarrow POSTOUTING$ route applied for both egress/ingress traffic.

3.3.3 Summary of Qualitative Comparison

Overall, the CNIs are plug-and-play components in Kubernetes. They provide an off-the-shelf network paradigm for the users to easily set up the networking environment. Based on Table. 3.1, we can observe that none of the existing CNIs cover all the qualitative features, *e.g.,* Flannel's primary drawback in lack of network policy support. Each CNI has its unique feature with the comparison to others. Users can choose the best-suited CNI in regard to their qualitative needs *e.g.,* network model, tunneling option, *etc.*

As shown in Table 3.1, Calico and Cilium use a Layer-3 based network model. Both of them support either an overlay or underlay solution. For the encapsulation options in overlay mode, Calico offers IP-in-IP and VxLAN while Cilium offers VxLAN and Geneve. Flannel, Weave, and Kube-router use a hybrid based datapath. Flannel and Kube-router support an overlay or underlay solution. Weave only supports an overlay. Flannel provides a simple network model, and users can easily set up an environment, while Weave and Cilium provide encryption support and also provide rich support for network policy and in addition Cilium and Calico are also able to support $3^{rd}party$ network policies and enable IPV6 support. And, Kube-router offers a unique DSR feature to enhance the load balancing in the Service-to-Pod mode of communication.

3.3.4 Additional Feature Considerations

(1) Multi-interface CNIs. Multi-interface CNIs are required in a number of cloud scenarios (*e.g.,* Virtual Private Network (VPN) connectivity, multi-tenant networks), where network isolation is needed [129]. By enabling multiple network devices and multiple subnets for a single Pod, Multi-interface CNI is able to provide a better separation of the control and user planes. A typical Multi-interface CNI is Multus [129]. Multus works as an orchestrator instead of configuring the Pod networking itself. It calls the Single-interface CNIs (*e.g.,* Flannel, Weave, *etc.*) to configure the underlying Pod networks, including routes, iptables, etc. With Multus, multiple Single-interface CNIs can coexist on the same host, and each Single-interface CNI has its own subnet, which is separated from each other. With the orchestration from Multus, we can choose the right subnet to use for each Pod based on its networking needs.

(2) Hardware Acceleration: Tunnel Offload. Most modern network interface cards (NICs) support the tunnel offload for a number of tunneling protocols, *e.g.,* IP-in-IP, GRE, VxLAN, GENEVE, *etc.* Tunnel offload includes the following components: TCP segmentation offload (TSO, which is also performed for normal IP), checksum processing, Receive Side Scaling (RSS) selection (for multiplexing/demultiplexing), *etc.* This can accelerate packet processing and reduce the load on the CPU. However, some NICs may only support a limited set of tunneling protocols to offload. This can severely impact the performance of the CNI plugin. For example, Flannel can be configured to use IP-in-IP or VxLAN tunneling for inter-host communication. Hence, it is necessary to make the appropriate choice of the CNIs overlay networking based on the hardware supported tunneling options.

We demonstrate the performance benefits achieved by the CNIs utilizing the tunnel offload capabilities. We find that it is necessary for the individual container framework to leverage the NICs offload capabilities and match it with the appropriate CNI to leverage that hardware acceleration to maximize performance.

3.4 Quantitative Evaluation and Analysis

In this section, we first compare the performance of different CNI plugins with a single connection between the Pods. Based on the performance results on the single connection, we breakdown the packet transmission overhead into different components and study the principal factors that influence the performance. Subsequently, we evaluate the performance with the increasing number of connections across Pods. Then, we examine the performance of different CNIs by applying a typical HTTP workload. We also assess the impact of netfilter rules and iptables chain configurations.

3.4.1 Experimental Setup

All the CNI plugins are evaluated on the Cloudlab testbed[191]. We build the Kubernetes cluster on two physical hosts. Each host machine has a Ten-core Intel E5-2640v4 at 2.4 GHz and 64GB memory, and 2 Dual-port Mellanox ConnectX-4 25 Gb NIC. We use Ubuntu 18.04 with kernel version 4.15.0-88-generic. Kubernetes is directly running on the physical machine, so there is no extra virtualization overhead introduced. All the Kubernetes related packages are installed with their current, latest version. We primarily use Netperf for throughput and latency measurement, with each test lasting 30 seconds and

being repeated 50 times. To fully utilize the bandwidth in both intra-host and inter-host communication, the application data size[16] in Netperf to 4MB for the TCP throughput test, which is the TCP socket buffer limit in the OS. For latency measurement, we use Netperf's request-response (RR) mode to get the round-trip time (RTT) for TCP traffic. The packet size used in the latency measurement is 1 Byte as a default. To examine the effect of Tunnel offload support, we choose another host machine with Tunnel offload supported on its NIC. This kind of host has two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz and 192GB Memory, and one Dual-port Intel X520-DA2 10Gb NIC.

3.4.2 Intra-Host Performance

We study intra-host performance (communication within a single server host) when communicating among a number of containers deployed within the host. This enables us to better understand and distinguish the communication overheads that arise due to the usage of the Linux bridge, iptables rules, eBPF, and the interaction with the host network stack. We evaluate Flannel, Weave, Cilium, Kube-router, and Calico variants (Calico-wp and Calico-np). 'Calico-np' helps to isolate the overheads of network policies (additional Netfilter rules).

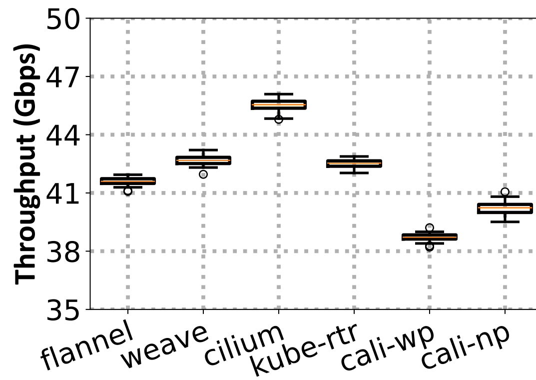
(1) Overall performance: The overall throughput comparison across CNIs is shown in Fig. 3.8(a). For TCP throughput, Cilium with its native solution based on eBPF outperforms the other alternatives. Layer-3 routing based solutions (Calico-wp and Calico-np) perform worse than the Layer-2 based solutions. Accordingly, we also observe that Cilium

[16] Application data size in Netperf is the "Send Message Size", the total data bytes to be transferred in a given connection, regardless of the socket buffer size, MSS or MTU settings.

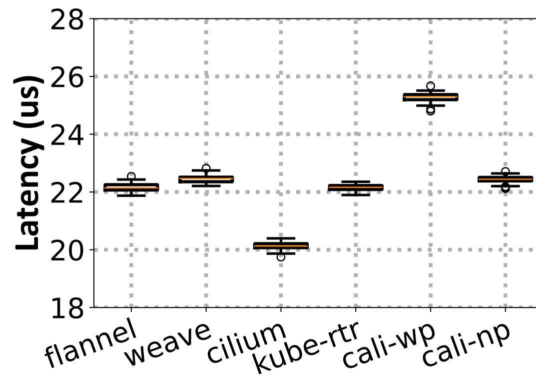
achieves the lowest latency and Calico-wp has the worst round-trip latency as shown in Fig. 3.8(b). This is primarily due to the overheads involved in processing the Netfilter rules and Layer-3 routing, which is avoided with the eBPF based CNIs. In order to understand how the datapath and iptables affect overall performance, we further break down the packet processing time into different components of the network stack and measure the CPU cycles as a packet goes through each component. We identify the following distinct components of the network stack: Forwarding Information Base (FIB), eBPF, Netfilter, Veth, and IP forwarding. By analyzing CPU cycles spent per packet in each component of the network stack, we establish the relationship between the achieved performance and the specific network activity involved in routing a packet with that specific CNI.

Methodology. In order to measure the *CPU cycles per packet* (CPP) spent in each network stack component, we first use the Linux *perf* tool [99] to count the total CPU cycles consumed in a 60-second packet transmission ($Cycle_{total}$) , which is repeated 5 times. We also use *perf* to trace the function calls and measure the percentage of the overall CPU cycles spent in the corresponding function ($Cycle_{percentage}$). With the total number of packets sent in a 60-second packet transmission (N_{packet}), we can calculate the CPP of a specific function call as follows:

$$CPP = Cycle_{total}/N_{packet} \times Cycle_{percentage} \tag{3.1}$$



(a) TCP Throughput



(b) TCP Round-trip latency

Figure 3.8: Intra-Host Throughput and Latency

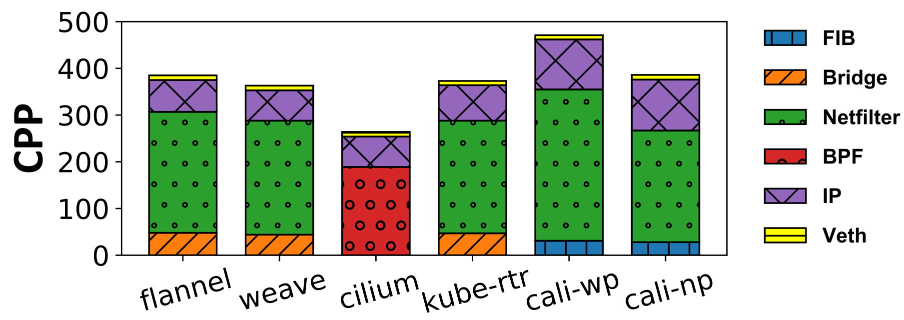


Figure 3.9: Overhead Breakdown for Intra-host scenario.

(2) Overhead breakdown for Intra-host Communication: The total CPP with the corresponding break down for intra-host communication is shown in Fig. 3.9. For the overall overhead of the complete network stack, Calico-wp (with Netfilter being the major contributor) has the highest CPP and Cilium the lowest.

Bridge: Flannel, Weave, and Kube-router use bridge-based solutions to forward packets in this intra-host scenario. When packets pass through the Linux bridge, the bridge-related function calls (*e.g., br_forward()*) are executed. Fig. 3.9 shows that the bridge overhead of Flannel, Weave, and Kube-router to be similar, at ~ 45 CPP.

FIB & IP forwarding: We put the FIB overhead and IP forwarding overhead together as 'IP forwarding' related function calls (e.g., *ip_forward()*) are coupled with FIB function calls (e.g. *fib_table_lookup()*). When using the host IP protocol stack to forward packets, first the FIB table lookup determines the next hop. Then, the packet forwarding operation is performed. As Calico relies on the host IP protocol stack to forward packets, it incurs both FIB and IP forwarding overheads. The FIB overhead of Calico (~ 30 CPP) is slightly lower than the overhead using the Linux bridge (~ 45 CPP). But, the IP forwarding processing in Calico consumes an extra 108 CPP. This overhead of both FIB and IP forwarding is higher than the overhead of the bridging approaches.

eBPF: Cilium relies heavily on eBPF. Instead of bridge/IP forwarding and Linux Netfilter, it utilizes a set of eBPF hooks in the network stack to run eBPF programs to support the intra-host packet forwarding and filtering functions. Cilium attaches the eBPF programs at each veth resulting in each packet forwarding operation to incur the eBPF processing overhead. Fig. 3.9 shows that Cilium has the eBPF overhead of ~ 189 CPP.

Veth: All CNI plugins spend almost the same ~ 10 CPP (for send, receive) on veth, a small percentage of the overall overhead, with little impact on the CNIs' performance differences.

Netfilter: Calico-wp, with calico policy fully installed, consumes 324 CPP on Netfilter, which is $1.35\times$ higher than the others. Although Calico-wp suffers a significant performance penalty due to the overhead from Netfilter, it allows better network policy customization and packet filtering due to fine-grained iptables chains. Calico-wp adds user-defined chains to construct its iptables, which introduces more iptables rules as well as more iptables overhead. Note: Cilium does not have any Netfilter overhead as it uses eBPF instead of iptables.

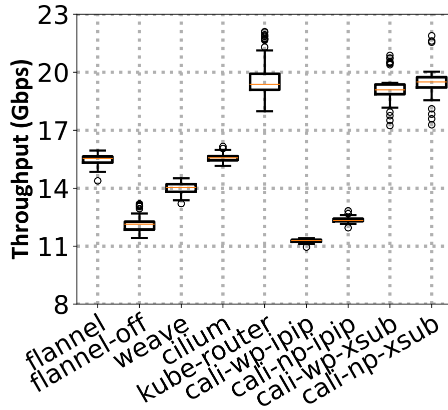
Summary: For intra-host communication, a native routing datapath based on eBPF is much cheaper than a bridge-based datapath or native routing datapath based on IP forwarding. eBPF combines packet forwarding and filtering together, which reduces the packet forwarding overhead. Thus, Cilium achieves the highest throughput and lowest latency. Further, a fine-grained iptables chain as in Calico-wp unfortunately hurts packet transmission performance. As shown in Fig. 3.8, Calico-wp has lower throughput and higher latency than the others, because of the penalty from the iptables chain processing.

3.4.3 Inter-Host Performance

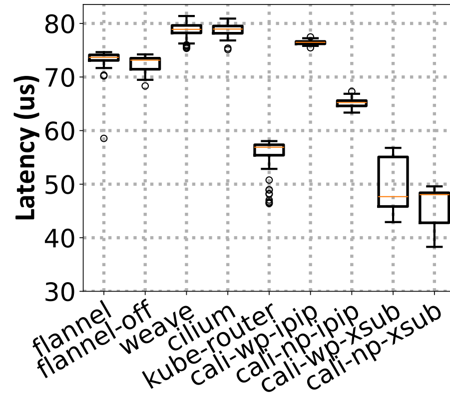
We use the same set of CNI plugins to communicate between two pods on different hosts. For all inter-host experiments, Flannel, Weave, and Cilium use the VxLAN overlay mode. Kube-router uses native IP routing (underlay), while Calico (wp or np options) can support either native IP routing (underlay) or IP-in-IP overlay. Accordingly for Calico,

we study all four modes: Calico-np-ipip, Calico-wp-ipip, Calico-np-xsub and Calico-wp-xsub. We noted that the offloading of IP-in-IP tunnel is not supported by the Mellanox ConnectX-4 25 Gbps NIC used in our testbed. Hence, to study the impact of tunnel offload in the NIC, we additionally experimented with Flannel by explicitly disabling the tunnel offload feature for VxLAN tunneling ('flannel-off' mode). We also study the inter-host communication performance on another Intel X520-DA2 10Gb NIC where IP-in-IP tunnel offload is supported.

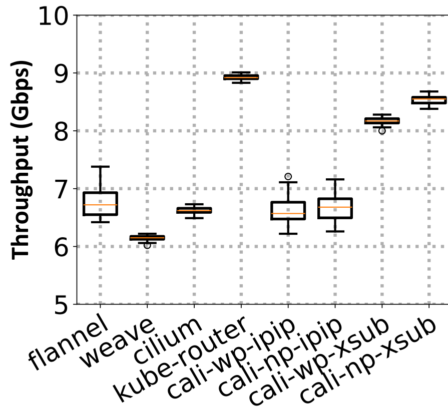
(1) Overall performance: TCP throughput and latency results are shown in Fig. 3.10. The native routing solutions (Kube-router, Calico-wp-xsub and Calico-np-xsub) perform better than the overlay solutions (Flannel, Flannel-off, Weave, Cilium, Calico-wp-ipip and Calico-np-ipip). However, without the tunnel offload in the Mellanox ConnectX-4 25Gb NIC, Flannel-off and Calico in IP-in-IP overlays perform poorly, with much lower throughput than Cilium and Calico with native routing (xsub) options. Moreover, for the same datapath, solutions with network policy disabled (Calico-np-xsub and Calico-np-ipip) perform $0.5 \sim 1.5 Gbps$ better than when network policy enabled (Calico-wp-xsub and Calico-wp-ipip). Also, TCP round-trip latencies show a corresponding increase for those CNIs that see lower throughput. For comparison purposes, we also show the results with Pods on two hosts communicating across a 10 Gbps link, using Intel X520-DA2 10Gb NICs that support both VxLAN and IP-in-IP tunnel offload. We see that the performance of Flannel and Calico IP-in-IP CNIs are comparable. In fact, because the CPU utilization is reduced due to the tunnel offload, the Calico CNI performance 'with policies' and 'no policy' see comparable throughput (however the latency is higher with policies). To sum-



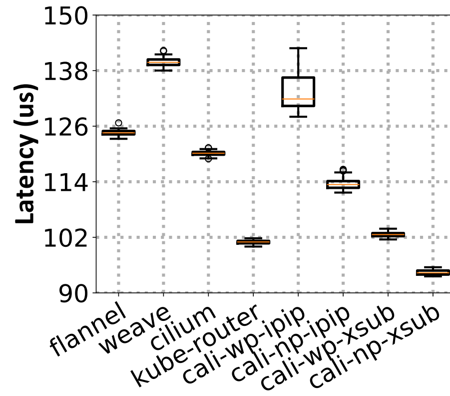
(a) Throughput (25Gbps NIC)



(b) RTT (25Gbps NIC)



(c) Throughput (10Gbps NIC)



(d) RTT (10Gbps NIC)

Figure 3.10: Inter-Host Throughput and Latency. (a) and (b) are with Mellanox ConnectX-4 25Gb NIC. (c) and (d) are with Intel X520-DA2 10Gb NIC.

marize, i) Tunnel offload is necessary to improve the CNI performance, and where the CNIs can support multiple overlay solutions (UDP, VxLAN, IP-in-IP), it is desirable to choose the overlay mode that can be supported via tunnel offload. ii) the underlay-based CNIs (Calico xsub and Kube-router) perform better than the overlay options, despite the NIC's offloading of the tunnel-related operations.

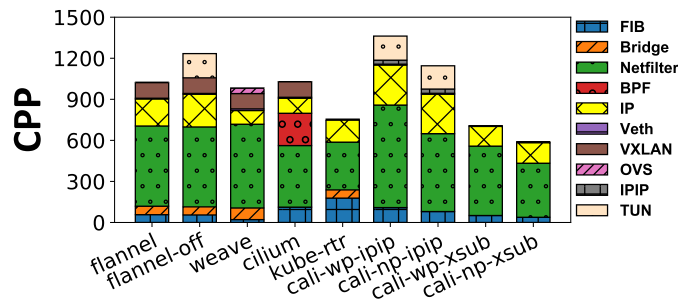


Figure 3.11: Overhead Breakdown for Inter-host Scenario: CPU overhead from both sender and receiver hosts

(2) Overhead breakdown for Inter-host Communication: For the overhead analysis, we include the VxLAN tunnel, IP-in-IP tunnel, OVS-datapath, and in-kernel tunnel offload processing components. Again, we use the same methodology to calculate the CPP of each function call on both the sender and receiver host. The total CPP, along with the breakdown is shown in Fig. 3.11. The native routing solutions (Kube-router, Calico-wp-xsub and Calico-np-xsub) have lower CPP compared to the overlay solutions (Flannel, Flannel-off, Weave, Cilium, Calico-wp-ipip and Calico-np-ipip). Also, the solutions with simple iptables have lower CPP than the complex iptables chains.

Layer-2 Bridging: Flannel, Weave, and Kube-router use the Linux bridge to forward packets between different virtual ethernet interfaces (veths) via Layer-2 bridging and incur

similar overheads as in the intra-host scenario. However, as Weave uses an extra veth-pair to connect the Linux-bridge with overlay tunnel, it incurs 2× more CPP than Flannel and Kube-router. With Weave, when the packets are forwarded from the veth to the overlay tunnel (sender side), they do not go through the host IP protocol stack but instead rely on the bridge-related function calls (i.e., *br_forward()*). These bridge-related function calls are invoked again when packets are forwarded from the overlay tunnel to the veth (receiver side). In contrast, Flannel and Kube-router use the host IP protocol stack to forward packets from the veth to the overlay tunnel, thus avoiding any bridge forwarding overhead on the sender side. On the receiving side, when forwarding packets from the overlay tunnel to the veth, they rely on Layer-2 bridge forwarding. Thus, the total bridge overhead of Flannel and Kube-router is half of that of Weave.

IP forwarding: Weave and Cilium traverse the host IP stack once per packet transmission and have similar ($100 \sim 110\,CPP$) overhead. The IP protocol stack operations of Weave and Cilium are performed between the VxLAN tunnel and host Ethernet interface. However, with Flannel, the host IP stack operations are performed twice per packet transmission. First between Linux bridge and VxLAN tunnel, and then again between the VxLAN tunnel and host Ethernet interface. Thus, we correspondingly incur about 197 CPP. Calico-*-ipip (Note: "*" means both Calico-wp-ipip and Calico-np-ipip) also performs host IP stack operations twice per packet transmission. First between veth and IP-in-IP tunnel, and then again between the IP-in-IP tunnel and host Ethernet interface. However, in this case, additional IP protocol stack processing overhead in the CPU occurs due to the lack of the IP-in-IP tunnel offload support in the NIC we used. This in-kernel tunnel processing (e.g., *ip_*

send_check() function call) expends more CPU cycles. Calico-*-ipip consumes $\sim 290\ CPP$ in the IP protocol stack. The underlay solutions (*i.e.,* Kube-router and Calico-*-xsub) consume $\sim 150\ CPP$ in IP protocol stack.

Netfilter: CNI's Network Policy is a useful feature that leverages Netfilter rules to enhance network security. However, Netfilter is a major source of overhead as shown in Fig. 3.11, requiring a larger amount of CPP than the other components. Large and complex iptables chains incur higher processing overheads. Fig. 3.11 shows that Cali-wp-ipip has the highest Netfilter overhead compared to the other solutions due to its large iptables size, while the Kube-router and Cali-np-xsub have the lowest. Cilium has the least Netfilter overhead compared to the other overlay-based solutions, as it bypasses the $PREROUTING \rightarrow FORWARD \rightarrow POSTROUTING$ route of Fig. 3.7 and uses eBPF instead.

eBPF: In the case of Cilium, we observe that the eBPF overhead for the inter-host packet forwarding is relatively higher (236 CPP) than the intra-host packet forwarding (189 CPP). This is due to an additional hook point at the host to support overlay mode and process the VxLAN tunneling.

Veth: As with the intra-host case, the veth processing overhead is the least compared to the other components and is similar for most of the CNIs ($\sim 6\ CPP$). As Weave has 4 veth-pairs on the inter-host datapath as opposed to 2 for the other CNIs, it incurs twice the overhead ($\sim 12\ CPP$).

Overlay: Using an overlay incurs packet encapsulation and decapsulation overheads. Flannel, Weave, and Cilium use VxLAN overlay and Calico-*-ipip use an IP-in-IP overlay. Overhead from the VxLAN overlay is $\sim 114\ CPP$, while IP-in-IP incurs somewhat lower

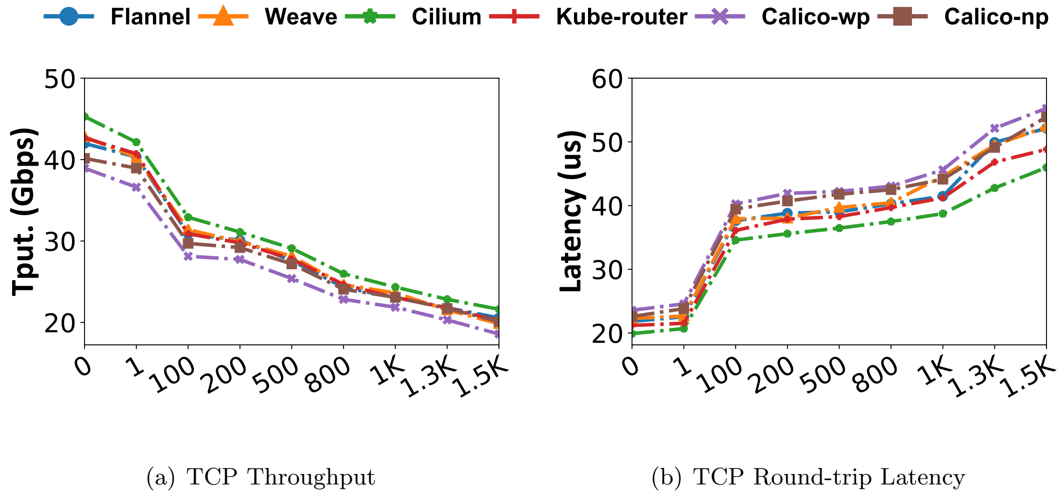
overhead ($\sim 30\ CPP$). Weave uses the OVS-datapath to implement the overlay processing and incurs some extra overhead ($\sim 40\ CPP$) on OVS-related function calls (Fig. 3.11). Thus, the overlay accounts for 3% $\sim$ 11% of the total overhead, depending on the packet encapsulation, potentially having a significant performance impact.

Tunnel Offload: The Mellanox ConnectX-4 NIC in our testbed machines support TSO and VxLAN offload, but not IP-in-IP tunneling[17]. Thus, all the IP-in-IP tunnel offload processing needs to be performed in the kernel, thus increasing the CPU burden. The in-kernel tunnel processing costs $\sim 170\ CPP$ ("TUN" in Fig. 3.11) for Flannel-off and Calico-*-ipip. As shown in Fig. 3.10, the Calico-wp-ipip and Calico-np-ipip only achieve 11.1*Gbps* and 12.5*Gbps* TCP throughput for inter-host pod communication respectively, which is much slower than the VxLAN overlay (14 $\sim$ 15.5*Gbps*) and Layer-3 routing (18 $\sim$ 19.7*Gbps*).

Flannel-off also shows the same significant performance reduction when tunnel offload is disabled on the NIC, confirming the extra processing overhead introduced in the kernel stack when the VxLAN processing is done in the CPU. In Netperf's request-response mode, the size of the payload is only one byte. The tunnel offload processing becomes important only when the size of the packet (including the tunnel header length) exceeds the Maximum Segment Size (MSS). So, with the small payload size, we see equivalent packet forwarding performance with/without hardware tunnel offload support. As shown in Fig. 3.10 (b), the Calico-wp-ipip has a similar RTT latency compared to some of the VxLAN overlay CNIs (*e.g.,* Weave, Cilium), while Calico-np-ipip achieves lower latency than the VxLAN overlay CNIs. Moreover, the Flannel and Flannel-off have similar RTT

[17] We verified using "ethtool" for the supported offload tunnel functions on the NIC. Also, refer to: (`https://community.mellanox.com/s/article/mellanox-adapters---comparison-table`).

latency, indicating that the effect of hardware tunnel offload support for small packets is not significant.



(a) TCP Throughput

(b) TCP Round-trip Latency

Figure 3.12: Intra-Host Performance with increasing # connections

Summary: Connecting the overlay tunnel and bridge via an extra veth-pair (as in Weave) may reduce the FIB and IP forwarding overhead, but increases the bridge and veth overhead. A powerful network policy mechanism can provide fine-grained packet filtering, allowing for improved security for packet transmission. However, more Netfilter calls result in lower packet forwarding performance. Users should carefully consider their needs for additional packet filtering rules and seek to manage the growth of iptables size as much as possible while meeting security requirements. Generally, a native routing datapath is cheaper than an overlay-based datapath. Removing unnecessary iptables chains and rules can help reduce Netfilter overhead. When used with the underlay network model, the Netfilter overhead of calico-wp-xsub can be kept around 507 CPP, which is lower than with the other overlay network models. Enabling a CNI's network policy support with native IP routing can

achieve a good balance between the overhead and network security support. A NIC's offload capability is another important aspect that needs to be seriously considered when choosing CNI. Having the kernel and CPU perform the tunneling tasks can have a measurable impact on performance.

3.4.4 Performance for larger-scale configurations

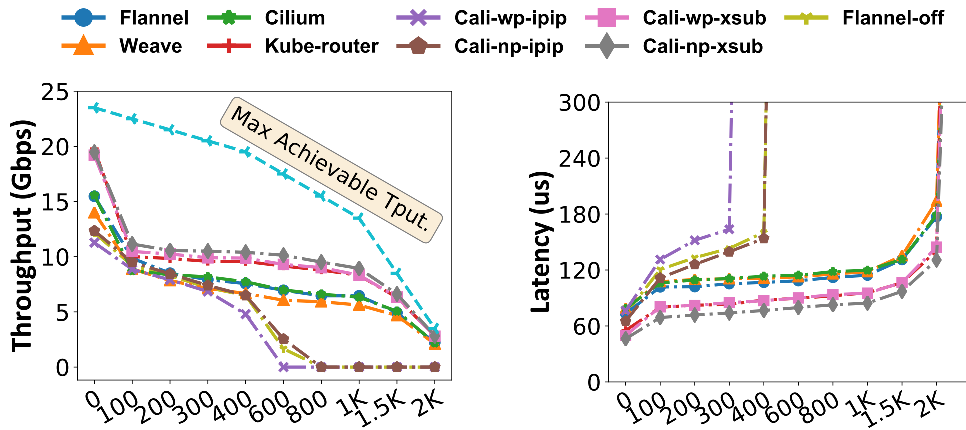
(1) Experimental setup: As the amount of communications between Pods increases, the individual conversation throughput will reduce, not only because of sharing resources among contending connections, but also interference and increasing overhead due to contention. To evaluate the performance variation with the scaling of the number of concurrent connections across Pods, we set up an increasing number of TCP connections between Pods as background traffic. The bandwidth of each of the background TCP connections was limited to 10Mbps for the inter-host case. As the intra-host has much higher achievable bandwidth, we set the bandwidth of each of the background TCP connections as 50Mbps for the intra-host case. We use *iperf3* to generate the background TCP connection traffic [9]. The test connection that we monitor generates traffic using Netperf in a distinct Pod communicating with a peer. Kubernetes has a maximum of 100 pods per host [2]. We have different levels on the number of background TCP connections in our experiments. For the intra-host case, we deploy 49 *iperf3* Pods each as the server and the client end on a single host. For the inter-host traffic, we deploy 99 *iperf3* Pods as servers on one host and deploy 99 *iperf3* Pods as clients on another host, each with 1 TCP connection. Two Pods are deployed for Netperf test for both scenarios. To generate background connections more than the number of *iperf3* Pods, we use "simultaneous connections" in *iperf3*.

(2) Intra-host performance with background traffic: Fig. 3.12 shows the intra-host performance comparison with different amounts of background traffic. The total background traffic with all 1.5K TCP connections (generating 50 Mbps each) is less than 75 Gbps. Cilium outperforms the other CNI plugins because of its low overhead in the datapath and in Netfilter. Calico-wp has the worst performance throughout, due to its large overhead from Netfilter rules. The TCP round-trip time is also better for Cilium. Further, we observe that even with just one active background connection, the test connection suffers about a $\sim 2Gbps$ throughput reduction. This drop is consistent across all the CNIs and happens as soon as two processes are involved in sending and receiving the packets on the same host. We speculate it to be likely a result of resource contention especially the locks.

(3) Inter-host performance with background traffic: Fig. 3.13 shows the inter-host communication performance (throughput and latency for a test connection) of different CNI plugins with varying amounts of background traffic, generated by an increasing number of background connections. The background connections not only consume bandwidth of the NIC and link, but also use up the host's CPU. However, the throughput reduction is more in line with the amount of added traffic from the background connections, except for the cases when the tunnel offloading is disabled, at higher loads. The difference among CNIs is noticeable in that the native routing solutions (Kube-router and Calico-xsub) outperform the overlay based solutions, because of less overhead on the datapath and Netfilter. In contrast, Calico in IP-in-IP mode performs worse than the others, because of the additional overhead since the NIC does not offload this function. This degraded performance is also observed with Flannel, when the tunnel-offload is turned off (Flannel-off). All of these see

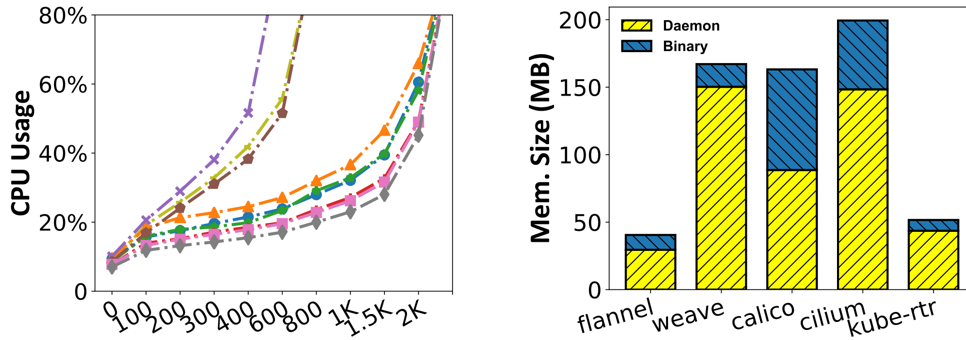
a precipitous drop in throughput beyond 400 background connections (each generating 10 Mbps) because the CPU is overloaded and the latency is correspondingly higher.

(4) CPU Utilization and Memory footprint: We also analyze the CPU utilization and memory footprint with different CNIs corresponding to these inter-host communication traffic experiments. Fig. 3.13(c), shows the CPU usage with the increasing workload. Native routing ('Calico-*-xsub', 'Kube-router') incurs relatively low CPU overhead, while the overlay mode CNIs (*e.g.,* Flannel, Weave, *etc.*) have a much higher CPU load. Further, CNIs with tunnel offload disabled are overloaded with fewer background connections, due to the extra offload processing in the kernel, which inevitably increases the CPU overhead. We also assess the memory footprint incurred by different CNIs in Fig 3.13(d). We observe that Flannel and Kube-router have a low memory footprint ($40 \sim 50\ MB$), while Weave, Cilium, and Calico have a very high memory footprint ($160 \sim 200\ MB$). We profiled the memory usage of CNIs 'daemonset' and 'binaries' from the kernel's pseudo-filesystem (procfs). We find the memory usage is independent of the number of pods/connections. The 'daemonset' is a running process with fixed memory size. And the CNI 'binary' file is an executable file, also with a fixed size.



(a) TCP Throughput

(b) TCP Round-trip Latency



(c) CPU Usage

(d) Mem. Size (MB)

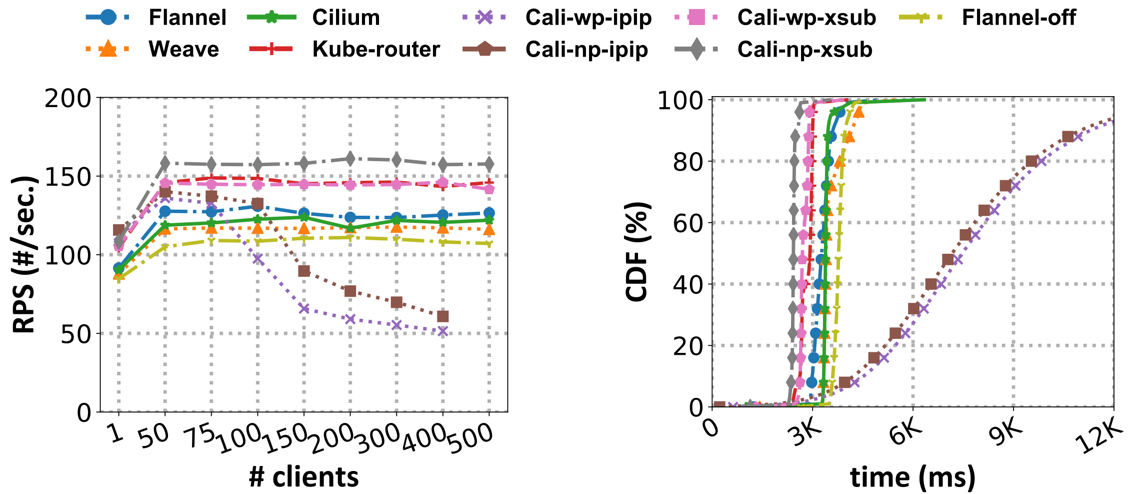
Figure 3.13: Inter-Host Performance with increasing # background connections; CPU and memory overheads

Summary: In general, increasing the amount of background connections impacts performance as they consume both the CPU resource and bandwidth. CNIs using native routing can achieve better performance compared to those using an overlay. Moreover, overlay CNIs without tunnel offload support in the NIC are impacted more, due to the increased in-kernel processing overhead.

3.4.5 Impact of CNI on typical HTTP workload

(1) Experimental setup: We use the Apache HTTP server benchmarking tool (*ab* [46]) to generate the HTTP workload. We emulate multiple concurrent clients and choose the *nginx* [12] as the HTTP server. We primarily study the CNI behavior in the inter-host scenario, deploying an HTTP server Pod and an '*ab*' client Pod on two different worker hosts. To get statistically reasonable results, we generate a total of 500K requests for each HTTP test. The size of the response payload for each request is fixed, at 5 MBytes. For each CNI, we also vary the level of concurrency (*i.e.,* number of clients sending HTTP requests simultaneously from 1, 50, 75, up to 500) to generate increasing amounts of HTTP traffic.

(2) HTTP Performance Results: Table 3.2 shows the performance with different CNIs. We can observe that Calico (both overlay and underlay modes) outperform the rest of the CNIs in a *single* connection case. However, with increasing numbers of connections ($c = 400$), Calico-*-xsub (underlay mode) works the best, while the Calico-*-ipip (overlay mode) turns to be the worst. This degradation in HTTP throughput is primarily due to the lack of tunnel offload support at the NIC, which results in high CPU overhead and thus



(a) RPS, increasing concurrency

(b) CDF Latency for concurrency 400

Figure 3.14: HTTP Performance for inter-host communication

adversely impacts the HTTP throughput and latency. Fig. 3.14 (a) shows the impact on RPS with different CNIs for the increasing number of concurrent connections and Fig. 3.14 (b) shows the latency profile with 400 concurrent connections. We can clearly observe that with the increasing number of concurrent connections (from 100 to 400) the performance of Calico-wp-ipip and Calico-np-ipip starts to degrade. We observed this to be due to additional CPU overhead generated by the IP-in-IP tunnel processing in the kernel. In fact, we noticed that the HTTP server often times out on the Calico-wp-ipip and Calico-np-ipip at a concurrency level of 500 (hence the numbers are not reported). The results indicate the overload behavior may be the cause of the poor performance for the Calico overlay CNIs [167]. From Table 3.2 and Fig. 3.14(b), we also observe that Calico-wp-ipip and Calico-np-ipip exhibit much higher average and tail latency than the other CNIs, for the case of 400 clients, suggesting much higher resource utilization on the server Pods.

Apart from Calico-*-xsub, we also observe that the underlay mode (Kube-router) performs somewhat better across all the cases when compared to the overlay modes (Flannel, Weave, and Cilium).

Summary Our realistic HTTP workload tests indicate there is a measurable impact of the choice of the CNI. In general, a 'Layer-3 + Underlay' CNI (*e.g.,* Calico, native routing) appears better suited for most HTTP traffic, especially at large scale (higher concurrency) traffic patterns.

Table 3.2: HTTP Performance of CNIs for 1 & 400 clients

CNI Metrics	**RPS**		**Avg.Latency (msec.)**		**Tput (Gbps)**	
	c = 1	***c = 400***	***c = 1***	***c = 400***	***c = 1***	***c = 400***
Flannel	91.50	125.23	10.930	3168.911	3.57	4.90
Flannel-off	84.66	108.15	11.812	3521.380	3.3	4.22
Weave	87.81	117.08	11.388	3362.784	3.43	4.57
Cilium	90.35	120.57	11.069	3217.960	3.52	4.70
Kube-rtr	104.50	143.38	9.569	2829.294	4.08	5.60
Cali-wp-ipip	112.20	51.46	7.987	7481.709	4.38	2.01
Cali-np-ipip	115.74	60.80	7.367	6629.865	4.52	2.37
Cali-wp-xsub	105.36	146.08	9.491	2738.195	4.11	5.70
Cali-np-xsub	108.31	157.28	9.233	2434.811	4.23	6.14

3.4.6 Iptables Evaluation

In order to study the impact of iptable rules (e.g., number of rules, etc) by different CNIs, we use the 'iptables-save' command to profile the iptable configurations set up by the CNIs. We track the chains and rules that a packet will traverse going from the source to destination Pod. Table. 3.3 presents the number of iptables chains and rules for different CNIs for the intra-/inter-host communication patterns respectively.

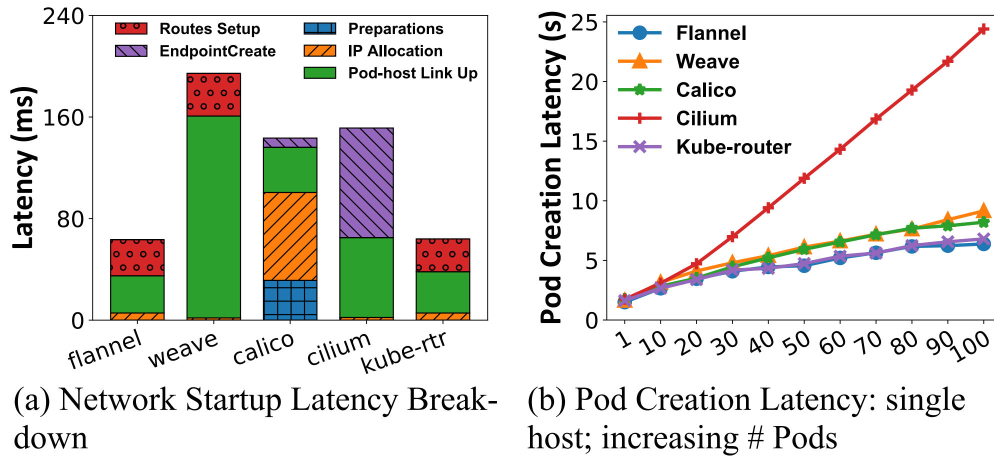
In the intra-host case, Cilium use eBPF for packet forwarding, which bypasses the iptables processing. Flannel, Weave, Kube-router, and Calico-np (both 'Calico-np-ipip' and 'Calico-np-xsub' in the intra-host case, as they are equivalent) have the same number of iptables chains and rules, and they all exhibit similar netfiler overhead ($\sim 245\ CPP$) for the intra-host case. Calico-wp (both 'Calico-wp-ipip' and 'Calico-wp-xsub') is configured with 17 iptables rules by default. This adds to a higher netfilter overhead ($324\ CPP$) compared to the others.

In the inter-host case, Calico with network policy enabled (*e.g.,* 'Calico-wp-ipip') has more iptables rules configured resulting in higher overhead ($749\ CPP$). Moreover, with a similar number of rules, a CNI with fewer iptables chains applied (*e.g.,* Cilium) has much less overhead ($450\ CPP$) than the CNIs with more iptables chains applied (*e.g.,* Flannel, Weave). We observed that CNIs using the underlay model have fewer iptables rules than the CNIs using the overlay model, as the overlay requires additional mangle/NAT rules to perform the necessary encapsulation/decapsulation. We also observed that the number of iptables chains and rules per packet does not increase with more background connections enabled.

Summary: Based on the iptables evaluation, we conclude that CNIs with fewer iptables chains and rules will have relatively less Netfilter overhead. However, to assure Pod network security, users needs to use the CNI's Network Policy API to install iptables rules, which could increase Netfilter overhead and lead to performance loss.

Table 3.3: Number of iptables chains and rules with default CNI configurations. Inter-host case collects from both the source host and destination host.

CNI Solutions	Intra-Host		Inter-Host	
	# of chains	*# of Rules*	*# of Chains*	*# of Rules*
Flannel	3	5	10	21
Weave	3	5	10	21
Cilium	0	0	4	20
Kube-router	3	5	6	10
Calico-wp-ipip	3	17	10	46
Calico-np-ipip	3	5	10	22
Calico-wp-xsub	3	17	6	28
Calico-np-xsub	3	5	6	12



(a) Network Startup Latency Breakdown

(b) Pod Creation Latency: single host; increasing # Pods

Figure 3.15: Pod Creation Latency with different CNIs.

3.4.7 Pod Creation Time Analysis

Starting a pod from scratch can take considerable time, and significantly impacts cloud-based microservices and Function as a Service offerings. The Pod creation latency is the time for starting a Pod from scratch until all the containers in the Pod have been created, comprising multiple steps [11], including network startup. We launch a single Pod each time and measure the network startup latency breakdown with different CNI plugins. We also measure the latency as we deploy an increasing number of Pods on a single host, with 10 repetitions. The container image used in our experiment is the 'pause container' [13], which just sleeps after being successfully started. The size of the pause container is 600KB.

The individual pod creation latency is in the range of $1.48 \sim 1.63s$ and the networking component contributes about $60 \sim 195ms$ (Fig. 3.15 (a)). Flannel and Kube-router have a smaller network startup latency ($\sim 60ms$) compared to the other alternatives. Weave consumes about $165ms$ in the *Pod-host Link Up* step, due to the work of appending multicast rule in iptables. Calico spends $\sim 80ms$ in the *IP Allocation* step, which is primarily due to the interaction with the etcd store. The time spent by Cilium in the *Endpoint Creation* step accounts for $\sim 90ms$. During this step, Cilium generates the eBPF code and links it into the kernel, which contributes to this high latency. Fig. 3.15 (b) shows the Pod creation latency for simultaneously starting a number of distinct Pods on the same host. Flannel and Kube-router have the smallest amount of increase in the creation latency as more Pods deployed together, while the latency with Cilium increases much more rapidly, which shows poor scalability.

3.5 Characteristics for an Ideal CNI

Based on our qualitative and quantitative analysis, we outline a design for an ideal CNI plugin and its characteristics. Our design choice for ideal CNI is motivated by the need to have very low overhead, low latency, and high throughput intra-host and inter-host container communication while also able to facilitate rich security and network policy support.

- Based on our evaluation results, we propose that an ideal CNI should seek to utilize the eBPF approach for *intra-host* communication. This is primarily because it generates the least amount of CPU overhead compared to the other solutions. We attach eBPF programs (with packet forwarding functionality) at the veth of Pods, so the intra-host packet forwarding can achieve better performance.

- For packet forwarding across hosts, we propose the ideal CNI use native (IP) routing. This helps avoid packet encapsulation/decapsulation overheads, avoids unnecessary fragmentation due to large MTUs, and results in fewer iptable-chains to process. This can achieve the highest packet forwarding performance when crossing the host boundary. The daemon of the ideal CNI needs to be able to configure BGP between nodes to distribute routing information, which is necessary to support native routing across hosts. Moreover, the host's physical interface needs to be attached with an eBPF program, so that we can leverage the benefits brought by eBPF, just as in the intra-host case. Multicast support can be built by leveraging eBPF's TC hooks, which can be a good match with the ideal CNI's intra-/inter-host datapath.

- When the underlying networking infrastructure does not provide support for native routing (such as BGP) or the users have a strong demand for network isolation, the ideal CNI should be able to offer sufficient overlay tunneling options to users (*e.g.,* IP-in-IP, VXLAN, GRE, *etc.*). Support for several overlay modes is desirable, especially if different NICs in the cluster lack the support for offloading certain specific overlay tunneling modes. A practical design should be able to auto-configure and suggest the right overlay option for the user. *I.e.* the CNI daemon should be able to detect the tunnel offload support information provided by NIC for all the nodes in the Kubernetes cluster (interact with 'ethtool'), and help users to make the right decision on choosing the overlay tunnel that can exploit the offloading capabilities, so as to achieve maximum performance.
- The ideal CNI should support the network policies that can be applied across all layers, *i.e.,* Layer 3 – Layer 7. This will provide a rich set of network policy attributes to provide needed network security. It is also desirable to have an eBPF-based iptables implementation [84], which could cooperate with the eBPF-based datapath design and achieve better packet filtering/forwarding performance.

3.6 Conclusion

Through qualitative analysis and a careful measurement-driven evaluation, we provide an in-depth understanding of the different CNI plugins, identify their key design considerations and associated performance. Our evaluation results show the interactions between the different datapath (organization of iptables), usage of the host network stack

contribute to the overall performance. While there is no single universally 'best' CNI plugin, there is a clear choice depending on the need for intra-host or inter-host Pod-to-Pod communication. For the intra-host case, Cilium appears best, with eBPF optimized for routing within a host. For the inter-host case, Kube-router and Calico are better due to the lighter-weight IP routing mode compared to their overlay counterparts. Although Netfilter rules incur overhead, their rich, fine-grained network policy and customization can enhance cluster security. Tunnel offload is another aspect to be considered, which can help to achieve the maximum performance when working with a CNI's overlay mode. This may be very desirable for Cloud Service Providers.

Our work sheds light on the benefits and overheads of the different aspects of CNIs, thus informing us of the design of an ideal CNI for Kubernetes cluster environments. The ideal CNI supports several desirable features including the eBPF-based intra-host datapath, native routing for inter-host packet forwarding, support for sufficient overlay tunneling options, automatic tunnel offload support detection, feature-rich network policy support coupled with an eBPF-based iptables implementation.

Chapter 4

An Efficient and Fair Algorithm for Serverless Function Placement

4.1 Introduction

Edge Clouds, aimed to place the data and applications closer to the user, are large numbers of ”tiny” data centers. Compared to today’s centralized data center, edge cloud data centers are scaled down and have a short distance from end users, which means much less transport cost and potentials for delay [16]. The prevalence of Internet of Things (IoTs) and latency sensitive cyber physical systems boost the demand for edge cloud computing, which is expected to serve enormous clients with high performance [190]. However, with the comparison of centralized cloud, edge cloud is far more resource limited [190], due to its small scale and dense multi-tenancy feature. An efficient, elastic and scalable resource utilization is the key to maximize the value of edge cloud. Serverless computing, driven by

the advancements of container technologies, has emerged as a promising paradigm to ease the concern on resource efficiency in edge cloud. Serverless computing liberates the users from managing the underlying infrastructure resources [77, 93]. The cloud service providers (CSPs) are in charge of provisioning, updating and managing the cloud resources. The resources can be scaled dynamically depending on the incoming load, which is advantageous to optimize the resource utilization and alleviate burdens on managing the workloads at scale. With serverless computing, the edge cloud is able to allocate resources on demand and hence improve the resource utilization.

Serverless function, which works as the Pod (i.e., a group of containers) under the context of Kubernetes-based platform, is a model for enabling convenient, on-demand access to a shared resource pool. It can be rapidly provisioned and released. The various types of hardware layer resources (e.g., CPU, memory, storage and network) can be abstracted and sold to client as functions. Resource allocation[1], as a vital aspect of both edge cloud and serverless computing, is still in face of several challenges: the increasing number of the clients and the heterogeneous resources. In order to improve the resource utilization of the serverless platform, the Pod placement should be optimized to fully utilize various types of resources and pack as many Pods as possible into the nodes, in order to avoid significant costs to start up new node instance. Besides, in consideration with user's experience of requesting microservices, a resource allocation decision, i.e., placing a requested Pod to its best-suited node, has to be completed within a limited time. Moreover, the well-known "cold start" problem can also benefit from efficient Pod placement. Thus, for the Pod placement in the context of serverless computing, the main optimization goal is to place

[1] "Resource Allocation", "Pod Placement" and "Pod Scheduling" are used interchangeably here

maximum number of Pods through a fast and efficient resource allocation scheme, which not only performs with low latency but also improves cloud's resource utilization.

In this chapter, we investigate in the optimal Pod placement scheme under multidimensional resource constraints, aimed to allocate the maximum number of Pods to a given set of nodes. We formulate the optimized Pod placement problem through a multidimensional bin packing model, with various resource types considered. In the future, a set of heuristic and approximation algorithms, e.g., relaxed solutions based algorithms, greedy algorithms, vector-packing algorithms will be examined on the proposed model. The best approach will be concluded based on the experimental analysis on the forementioned algorithms. We also plan to apply the proposed model to the popular open-source serverless platform — Knative — to study its resource utilization improvement under a realistic edge cloud environment.

Placement Engine (§4.3): Mu carefully decides where to place each function container to avoid fragmentation and ensure fairness in a multi-tenant environment. Unlike centralized clouds where resources are seemingly endless, Edge clouds may frequently run at close to capacity. We show our placement heuristic achieves comparable performance to optimization techniques at a much lower computation cost and provides up to a 2× improvement in fairness among tenants.

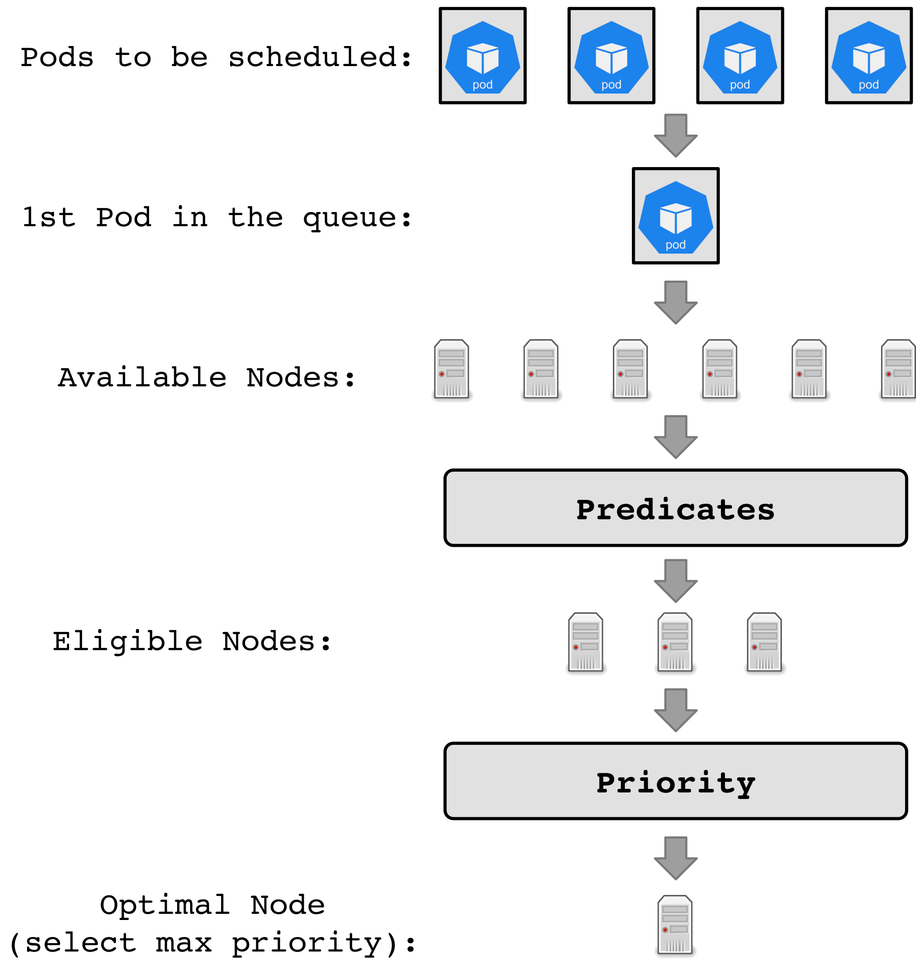


Figure 4.1: The scheduling model of Kubernetes

4.2 Background

4.2.1 Kubernetes Scheduler (Placement Engine)

The scheduling model of Kubernetes is shown in Fig. 4.1. Once a new Pod is created, either by auto-scaler or user manually, it will be inserted into the waiting queue in the scheduler before being deployed to a suitable node. During each scheduling cycle, only the 1st Pod in the waiting queue can be scheduled. The remaining Pods will stay in the queue and wait for the next scheduling cycle. The scheduling decision making process consists of two steps: predicate and priority. In the predicate step, the scheduler will filter out the ineligible nodes, which fail to meet the specific resource requirements (e.g., CPU, memory, etc) of the Pod. A set of filtering plugins has already been offered by Kubernetes and can be applied in the predicate step, e.g., *PodFitsResources*, *MatchNodeSelector*, etc. In some corner cases, there would be no eligible node to be deployed with the target Pod after the predicate process. The scheduler will set the status of the Pod as failed deployment, report to the user and then start the next scheduling cycle.

In the priority step, there would be several eligible nodes retrieved after the predicate step. Scheduler will prioritize all the eligible nodes by using priority plugins provided by Kubernetes. Various priority plugins can be applied in this step to find the best-suited node to provision the Pod, e.g., *LeastRequestedPriority*, *MostRequestedPriority*, etc. After the completion of the priority process, it is possible to find more than one node is scored as highest priority. In this case, the scheduler will randomly select one of them. With the scheduling decision on the given Pod, the scheduler will inform the kubelet on the selected node through the API server, and then the kubelet will bind the Pod to the selected node.

To help the scheduler makes better scheduling decision, users can specify the resource requests and limits in the Pod's configuration. The resource request determines the minimum resource requirements of the Pod and the resource limit indicates the maximum resources that the Pod can acquire. Based on the configuration of resource requests & limits, Pod can be classified into various QoS classes, i.e., *Best Effort*, *Burstable* and *Guaranteed*. The *Best Effort* Pod has no configuration or guarantee on its resource requests & limits, which means if the node exhausts its resources, the *Best Effort* Pod will be evicted first. If the resource requests of the Pod is configured less than its resource limits, it will be classified as *Burstable* Pod. For the *Burstable* Pod, it can use amount of resources more than requested if there are remaining resources available on the node. The resource requests of the *Guaranteed* Pod is configured to be equal to the resource limits, which means the *Guaranteed* Pod is able to use maximum resource that allowed by the node. For the sake of better utilizing node resources as well as ensuring good performance, the users should properly setup the resource requests and limits of the Pod. On the one hand, unexpected resource contention between different Pods will definitely degrade the overall performance. On the other hand, overestimated resource configurations will lead to wasted resources, as the Pod cannot use up the entire amount of allocated resources.

4.2.2 Knative Serverless Platform

Knative is a well-known open-source serverless platform, which is applicable for deploying and serving serverless applications. Knative runs on top of Kubernetes. Equipped with a set of Kubernetes-based middleware components, Knative offers the ability to ex-

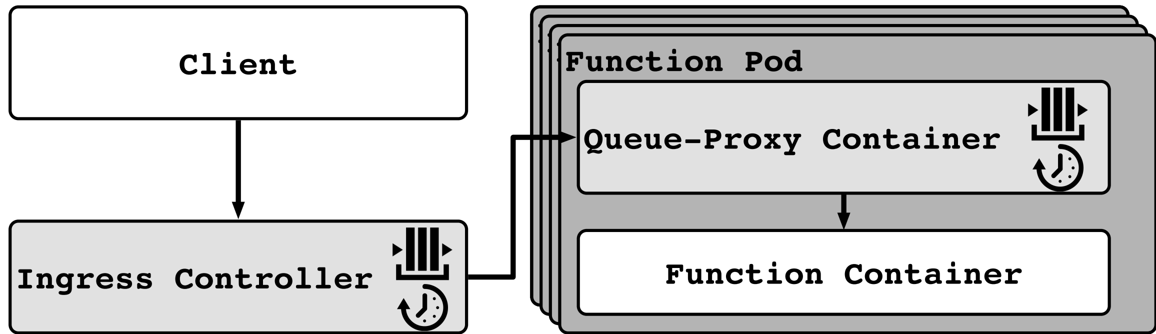


Figure 4.2: The working model of Knative platform

ecute FaaS workloads that can be elastically scaled [59]. The working model for Knative is shown in Fig. 4.2 [151]. The requests generated by the external client will be queued by ingress controller (from 3rd party, e.g. Istio ingress controller) and wait to be fetched by the function Pods. Two containers, the queue-proxy container and the function container, constitute the function Pod. When the requests reach the function Pod, they will firstly be queued in the queue-proxy container, and subsequently, forwarded to the function container for further processing. The queue-proxy container enables concurrent requests to be processed by the function container. The users can specify the concurrency level in accordance with their needs [10]. Compared to other alternative serverless platforms, e.g., Nuclio, OpenFaas, Knative suffers from higher communication overhead resulted from its dual-container implementation, which indicated worse performance [151].

4.3 Placement Engine Design

Fig. 4.3 shows the architecture of Mu, which builds on the Knative, Kubernetes, and Istio tools. The Placement Engine must pack pods to suitable nodes to reduce resource

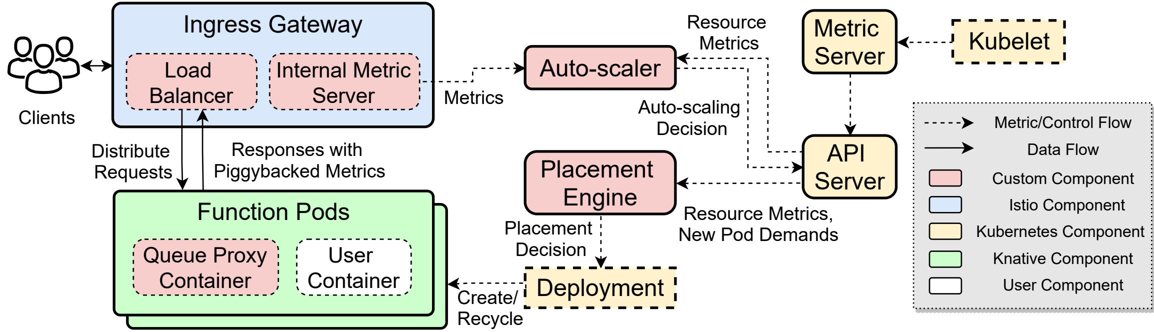


Figure 4.3: Mu Overview.

fragmentation, improve the efficiency and ensure fairness between functions when Edge resources are constrained.

When functions have to be instantiated, the typical approach in Kubernetes and Knative is to use a bin-packing algorithm to schedule (place) the function pods on available servers. We develop an efficient and fair algorithm for a placement engine to pack function pods to suitable nodes while reducing resource fragmentation. Since an edge cloud may have limited resources, it is important to fairly allocate resources among contending functions, while considering their demand for resources across multiple dimensions (CPU, memory, *etc*). We adapt the notion of dominant resource fairness (DRF) to arrive at a fair placement strategy [113].

4.3.1 Optimization Model and Metrics

We first model the function placement task as an Integer Linear Program (ILP) formulation. Let N be a set of nodes; Let J be a set of resources; each resource $j \in J$ has its capacity $c_{n,j}$ on node n. Let F be a set of functions, each function $f \in F$ has its desired pod count p_f. Each function's pods demand $d_{f,j} \geq 0$ on resource j, and $w_{f,n}$ denotes the

number of function f's pods placed at node n. We define two objective functions for two alternate models, `ILP0` and `ILP1`, both of which have the same constraints, as below:

$$\begin{aligned} \texttt{ILP0:}\max \quad & \textstyle\sum_{n\in N}\sum_{f\in F} w_{f,n} \\ \texttt{ILP1:}\max \quad & \textstyle\sum_{n\in N}\sum_{f\in F} \frac{1}{D_f} \times \sum_{i=1}^{w_{f,n}} \frac{1}{i} \\ \text{s. t.} \quad & \textstyle\sum_{f\in F} d_{f,j} \cdot w_{f,n} \leq c_{n,j}, \forall n \in N, \forall j \in J \\ & 0 \leq \textstyle\sum_{n\in N} w_{f,n} \leq p_f, \forall n \in N, \forall f \in F \end{aligned} \tag{4.1}$$

The goal of `ILP0` is to maximize the total number of pods and thus the overall resource efficiency among a given set of nodes, while `ILP1` maximizes both the resource efficiency and fairness across different functions. `ILP1` assigns a weight $w_{f,n}$ by decreasing the reward for placing a function's as the number of pods increases for that function. Thus, the reward for placing more pods for a single function is less than the reward of evenly placing the pods of several different functions. In addition, to ensure the function with a small resource demand will not be starved by functions with a large resource demand, `ILP1` weights the $w_{f,n}$ by the dominant resource share of function f ($D_f = \max_{j\in J} \frac{d_{f,j} \times p_f}{\sum_{n\in N} c_{n,j}}$). Placing a large function pod receives a smaller reward, which guarantees fairness between large functions and small functions. Both `ILP0` and `ILP1` are constrained by the node's resource capacity and each function's requested pod count.

Quantifying Fairness & Efficiency: We quantify fairness of the allocation of resources to each function by the placement engine based on the principle of Max-Min fairness [85]. With varying demand from contending functions, it is important to evaluate fairness also as a function of time. Similarly, we evaluate the efficiency of the placement engine, by comparing its allocation with an allocation that maximizes the resource efficiency, as spec-

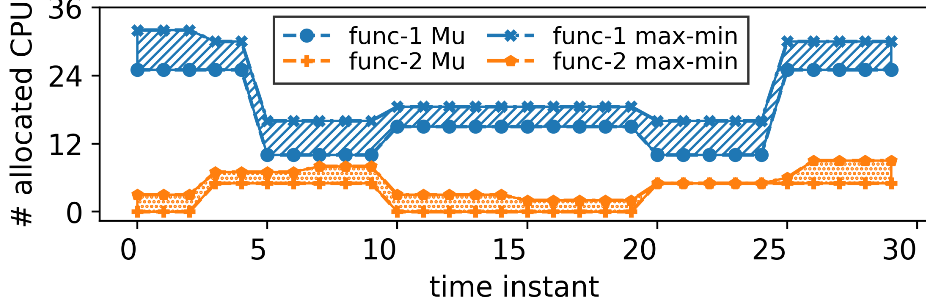


Figure 4.4: An example on integrating the degree of CPU unfairness over 30 intervals

ified by the greedy algorithm `ILPO` above. We evaluate the degree of unfairness U_j and the inefficiency I_j of a placement algorithm on resource j integrated over a period of time, T, in Eq 4.2.

$$\begin{aligned} U_j &= \frac{\sum_{f \in F_t} \sum_{t \in T} |R_{f,j,t} - M_{f,j,t}|}{\sum_{t \in T} |F_t|} \\ I_j &= \frac{\sum_{t \in T} |\sum_{f \in F_t} d_{f,j,t} - \sum_{f \in F_t} R_{f,j,t}|}{\sum_{t \in T} |F_t|} \end{aligned} \tag{4.2}$$

where $M_{f,j,t}$ indicates the max-min allocation on resource j to function f at time instant t, and $R_{f,j,t}$ is the the amount of resource j allocated to each function f by the placement algorithm at time t. Ideally, a placement algorithm could directly meet the max-min allocation, but in practice this is not possible because it only considers a single resource and assumes resources can be allocated without any fragmentation. Fig. 4.4 shows an example on quantifying the degree of unfairness, by comparing the allocations to two functions over time with regard to their ideal max-min allocation. We integrate the absolute difference between $R_{f,j,t}$ and $M_{f,j,t}$ over a period of time T, $(\sum_{t \in T} |R_{f,j,t} - M_{f,j,t}|)$. The degree of unfairness can then be calculated by averaging the cumulative area of all the functions over the entire time period T. Since the max-min allocation achieves the optimal fairness for each resource [85], a larger U_j indicates more unfair allocation on resource j. We do the same

for the degree of inefficiency, integrated over time to get the overall degree of inefficiency, I_j, of the placement algorithm compared to the placement with the greedy algorithm `ILP0`.

Algorithm 1 Placement Algorithm

```
1: while F ≠ ∅ do
2:     D_f ← {max_{j∈J}(R_{f,j} / Σ_{n∈N} c_{n,j}) | ∀f ∈ F}
3:     Pick the function f' ∈ F with minimum D_f
4:     S_n ← {score_{n,f'} | ∀n ∈ N, d_{f',j} fits in a_{n,j}}
5:     if ∀n ∈ N, S_n = φ then
6:         F ← F − f'
7:     else
8:         Place f' to node k ← max_{n∈N} S_n
9:         R_{f',j} ← R_{f',j} + d_{f',j}; a_{k,j} ← a_{k,j} − d_{f',j}; p_{f'} ← p_{f'} − 1
10:        if p_{f'} = 0 then
11:            F ← F − f'
12:        end if
13:    end if
14: end while
```

4.3.2 Heuristic algorithms

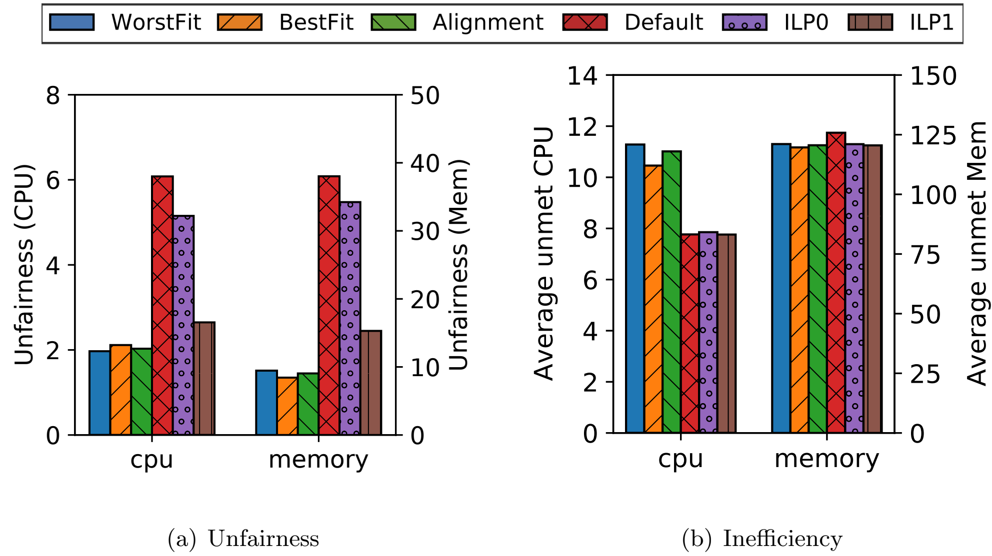
As the ILP model is NP-Hard, we also design a heuristic algorithm to solve the pod placement. We break the placement algorithm into two modules: *(i)* the pod selection module considering Dominant Resource Fairness (DRF), to decide which function pod is selected to be placed next; *(ii)* the node selection, which chooses the node to place the

function pod at, based on a scoring function, so as to reduce the resource fragmentation, while minimizing unfairness. For a given scaling decision from Autoscaler, the placement engine invokes these two modules iteratively until function set F is empty (Algorithm 1).

Module 1: Pod selection. We calculate dominant share (D_f) of every function and pick the function f' with the minimum D_f. If multiple functions have the same minimum D_f, the function with the minimum sum of the resource demands (*i.e.,* $MIN(\sum_{j \in J} d_{f,j})$) is selected.

Module 2: Node selection. We evaluate a number of existing scoring functions for selecting the node, *e.g.,* `Alignment` [117], `WorstFit` [177], and `BestFit` [177]. `Alignment` uses $score_{n,f} = \sum_{j \in J} \frac{a_{n,j}}{c_{n,j}} \times \frac{d_{f,j}}{c_{n,j}}$ to score the node n for the selected function f, $a_{n,j}$ is the remaining resources on node n. `Alignment` picks the node with the highest amount of remaining resources. `WorstFit` chooses the node with the highest value of: $\sum_{j \in J} \frac{a_{n,j} - d_{f,j}}{c_{n,j}}$. Thus, `WorstFit` seeks to pack the function into the node with the least amount of resources available that can accommodate this function's demand. `BestFit` chooses the node that has the highest value of: $\sum_{j \in J} \frac{a_{n,j} - d_{f,j}}{a_{n,j}}$. `BestFit` seeks to pack the function into the node with the most amount of resources left after accommodating this function's demand. All nodes $n \in N$ that have enough resources to fit the selected function f', are then scored using a scoring function. If f' has no node with a valid score, it is removed from F. Else, the node with maximum score is picked for f'. After placing f', we update the resource allocation and capacity. If the total pods demanded for function f' is met, it is removed from the set F.

4.4 Placement engine evaluation



(a) Unfairness

(b) Inefficiency

Figure 4.5: Fairness and efficiency comparison (simulation)

4.4.1 Simulation

We simulate and compare our different DRF-based heuristic approaches (`WorstFit`, `BestFit`, and `Alignment`), the default Kubernetes scheduling heuristic (`Default`), and the two ILP models, by setting up 500 randomly generated placement test cases. We consider a simulated cluster of 40 nodes and 300 functions. A workload generator is used to randomize functions and nodes in each test. In the configuration used, 90% of the functions require less than 400MB memory while 10% of functions require 500∼2000MB memory. The CPU demand of functions ranges from 1∼8 cores. Each function requests 1∼16 pods. To ensure demand exceeds the resources in the cluster, the total CPU capacity of the cluster is set to 80% of the total CPU demand and the total memory capacity is set to 60% of the total

memory demand. We use Gurobi [119] to solve the ILP models, adjusting the accuracy and termination criterion to keep computation time manageable.

Fig. 4.5(a) shows the fairness (as defined in Eq. 4.2) of the allocation decisions for the CPU and memory. The 3 DRF heuristic-based algorithms (which are all close to each other) achieve 2× better fairness than the ILP0, which does not consider fairness in its optimization. `ILP1` considers the fairness in the formulation, and achieves better fairness than the `ILP0`. However, with the accuracy and termination criteria we used with the solver, `ILP1` achieves better efficiency but poorer fairness than the DRF heuristic algorithms. The worst algorithm in terms of fairness is the Kubernetes `Default` approach. Comparing CPU efficiency (Fig. 4.5(b)), the DRF heuristics have an unmet CPU demand of ∼10 cores on average, which is slightly worse than the ILP models. The Kubernetes `Default` is also better, with an average unmet CPU demand of ∼8 cores. All the alternatives have similar memory efficiency, resulting in an average of $\sim 130MB$ unmet memory demand. Thus, the DRF heuristic approaches strike a good balance of having very good fairness, and are close to the best case efficiency of the Kubernetes `Default` algorithm (which however ignores fairness). In Mu's deployment, the placement engine is executed once every epoch (2 seconds, driven by the autoscaler). In terms of computation time, the `Default` Kubernetes approach takes ∼200 ms. The DRF heuristics are also fast, taking ∼500 ms to determine the placement of 300 functions among 40 nodes. However, the ILP models, depending on the accuracy desired, take much more time ($>$ 2 seconds on a server-class machine) and are impractical for real-time placement use. The DRF approaches, on the other hand, are feasible for deployment.

4.4.2 Evaluation of Mu's Placement Engine with Real Workloads

Table 4.1: Experiment configuration

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

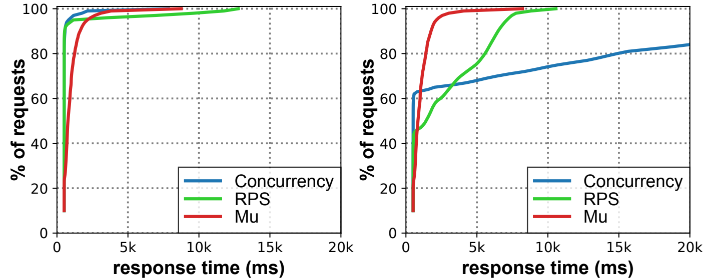


Figure 4.6: Response time CDF for 3 frameworks for Workload 1 (left); Workload 2 (right; only partial CDF for Concurrency)

We now evaluate Mu's placement engine for a few large scale workloads. We compare Mu with the Knative default approaches.

Implementation Details and Testbed Setup: Mu's implementation extends multiple components in the Knative ecosystem, including the Knative Queue-Proxy, Istio Gateway, Knative Autoscaler, and Kubernetes Scheduler (placement engine). We base our code on Kubernetes v1.17.0, Istio's Envoy Proxy v1.16.0, and Knative v0.13.0. Our extensions comprise ∼800 lines for the placement engine. We evaluate the serverless platforms on the Cloudlab testbed [102] consisting of one master and ten worker nodes, each of them equipped with Two Intel E5-2660 v3 10-core CPUs at 2.60 GHz (40 hyperthreads per host) and 160 GB ECC memory running Ubuntu 18.04.1 LTS. We do not add any extra pod heterogeneity in this experiment other than the natural fluctuations found on CloudLab.

To comprehensively evaluate Mu, we use the workloads received by functions in the Azure dataset [196]. We select 2 workloads with variable invocation patterns from

the top 10 workloads sorted by maximum number of invocations for the first day in the dataset. We scale down these workloads by dividing the number of invocations by 100 for the experiment, treating each minute of the original trace as one second to add dynamics. The scaled down workload and the configuration of the serverless environment are in Table. 4.1. With the fairness-aware Placement Engine, Mu can more fairly allocate the limited edge cloud resources among the competing functions.

Table 4.2: Comparing Mu with the standard Knative build

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

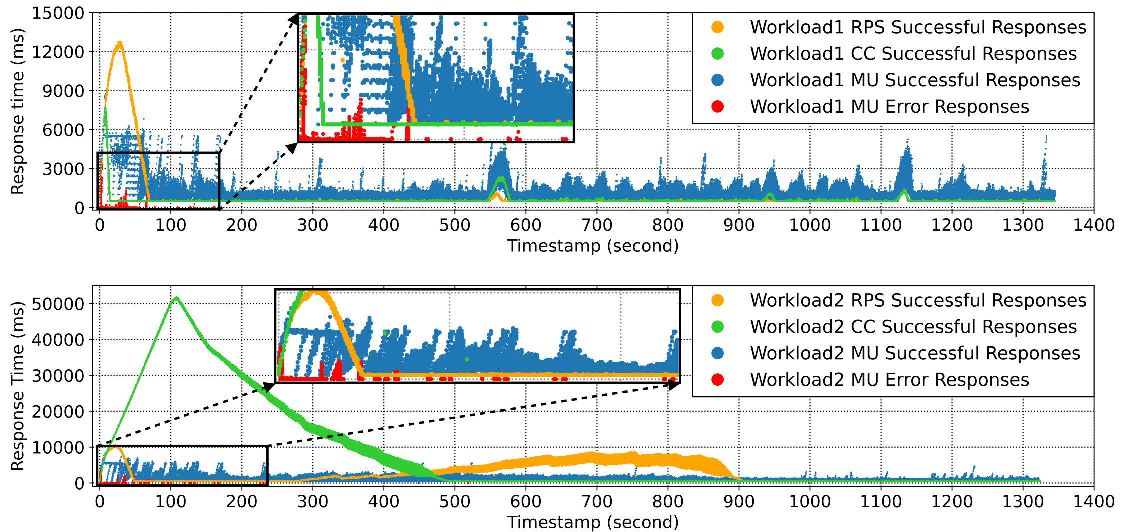


Figure 4.7: Time series of Response Time for Mu, RPS, and Concurrency (Top: Workload 1; Bottom: Workload 2)

Latency and Fairness: The CDF of the response times for each workload and approach is shown in Fig. 4.6. Mu has good control over the response times and limits the tail latency that exceeds the specified SLO of 5 seconds for both workloads. For Workload 2, Mu provides a substantially tighter response time distribution than RPS or Concurrency. As shown in Table 4.2, the 99% response time for the two workloads are both below the 5 second SLO for Mu. Examining the response time distribution (Fig. 4.6), and the average and 99%iles (Table 4.2) and the time series of the response times (Fig. 4.7(a), 4.7(b)), we see that Mu maintains fairness between the workloads for the entire length of the experiment.

In contrast, the standard Knative approaches result in much larger response time tails, and both unfairly treat one of the workloads. For Workload 1, both RPS and Concurrency (CC) achieve a lower average response time (except RPS has a relatively large number of requests experiencing high delays at the start of the workload, resulting in its 99%ile being higher). However, for Workload 2, both RPS and CC behave quite poorly at different periods of the workload execution, as seen from the time series (Fig. 4.7(b)), with 25-32% of requests violating the SLO. Workload 2 sees an unacceptably large 99% latency with CC as seen in Table 4.2. Since Mu is conservative in its pod allocation for both Workload 1 and 2, it sees a slightly higher average response time for Workload 1 than RPS and CC, but better for Workload 2 than RPS and CC. The 99%ile for Mu is clearly better than the two alternatives.

Pod Allocations: We use the term "requested pod count" for all alternatives. It comes directly from the autoscalers for RPS/CC. For Mu, the placement engine uses the pod count determined by the autoscaler and accounts for fairness and overall system capacity

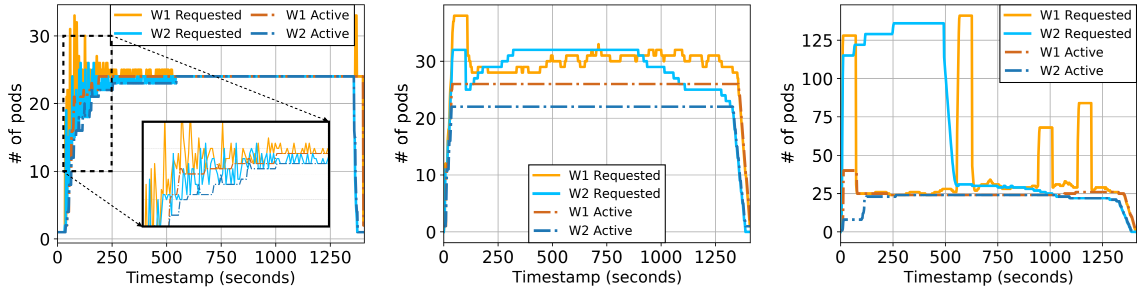


Figure 4.8: Time series of Pod counts for Mu (left), RPS (middle), and Concurrency (right)

to determine Mu's "requested pod count". On average RPS and Concurrency use 18% and 17% more pods than Mu. Mu tends to request fewer pods since its goal is to proactively provision enough pods to meet SLOs, and the predictor helps to anticipate the future workload. In Fig. 4.8(a), Mu is aware of both the workloads and fairly determines the requested pod count. RPS and concurrency on the other hand (Fig. 4.8(b) and 4.8(c)), run an autoscaler for each workload, without coordination between decisions for each workload. Thus, the requested pod counts may not only be unattainable, but can also be unfair. This is most evident for concurrency based autoscaling in Fig. 4.8(c) where the pod requests for individual workloads exceed 100, whereas the system capacity only allows provisioning 48 pods totally.

SLO Performance: Overall, Mu provides a significant increase in the total number of requests served within the SLO (96.8%) compared to the RPS scaling policy (86.2%) and Concurrency scaling policy (84.2%), as shown in Table 4.2. Mu uses SLO-aware admission control and returns 503 errors [19] for requests which it will not be able to serve within the SLO based on current queue lengths. This avoids the build up of a large queue with the arrival of a burst of requests. RPS and concurrency do not factor SLO into account, so

when bursts occur, requests are buffered in the activator, and the queuing results in a large number of SLO misses. Throughout the experiment, Mu has relatively uniform response times, increasing only during bursts, when the system is under-provisioned (e.g., first 200 seconds of the experiment when we have to scale up from zero to a large number (~ 20) of pods). On the other hand, Concurrency and RPS see persistent queuing for long periods (> 400 seconds) and the response time grows substantially more than the desired target SLO of 5 seconds. There is also significant unfairness for Workload 1 vs. Workload 2 as seen in Fig. 4.7(a), 4.7(b).

As shown in Fig. 4.7, Mu returns 503 errors (indicated by red dots). Our view is that by having these failures (and potentially having those requests be retransmitted) impacts a relatively small number (less than 5%) of requests, which is better than building up a large queue resulting in very long latencies for a large number of requests (25-30%, as seen for RPS and Concurrency) and likely to more seriously impact user Quality of Experience (QoE). These 503 errors are well correlated with the occurrence of bursts when resources are not yet provisioned by Kubernetes. This is mitigated somewhat by the predictor and proactive autoscaling. In fact, most of the 503 errors occur when the burst arrives at the beginning when the predictor has not yet learned the characteristics of the workload. Additionally, even though Mu's autoscaler requests allocation of a larger number of pods, Kubernetes can take a large amount of time to provision these pods, starting from an initial zero-scale system (as seen in the difference between pods being requested and active in the first 200 seconds for Mu (see Fig. 4.8(a)).

4.5 Conclusion

Existing platforms such as Knative suffer from their ad-hoc design that leverages existing frameworks such as Kubernetes without substantial customization for serverless use cases (*e.g.,* reuse the Kubernetes placement algorithm). Further, today's serverless platforms are designed for large scale cloud environments with abundant resources, without meeting the strict requirements of agility and efficiency needed for Edge cloud environments.

Our work on Mu demonstrates the importance of a fairness-aware placement engine. When resources become overcommitted, Mu's placement engine ensures greedy functions cannot unfairly starve others. We have demonstrated that Mu's placement engine improves fairness for Edge environments, leading to more consistent performance and fairness across functions, while avoiding long tails for the response time.

Chapter 5

High-performance and Efficient Serverless Computing using Event-driven Shared Memory Processing

5.1 Introduction

Serverless computing has grown in popularity because users have to only develop their applications while depending on a cloud service provider to be responsible for managing the underlying operating system and hardware infrastructure. The typical costs borne by the user of serverless computing are only for processing incoming requests. This **event-driven** consumption of resources is attractive for cloud users, especially when their demand

is intermittent. It does, however, place the burden on the cloud service provider to provide adequate resources on-demand and ensure the quality of service (QoS) requirements are met.

In many cases, serverless frameworks are profligate in their resource consumption. They provide the needed functionality by loosely coupling serverless functions and middleware components that run as a separate container and/or pod. This can be extremely resource-intensive, especially when deployed in a limited capacity environment, *e.g.,* edge cloud [166]. There are still a number of shortcomings to be overcome for building a high-performance, resource-efficient, and responsive serverless cloud. Some contributors to this overhead are the following.

Use of heavyweight serverless components. In a serverless environment, each function pod has a dedicated sidecar proxy, distinct from its application container. Sidecar proxies help build an inter-function service mesh layer with extensive functionality support, *e.g.,* metrics collection and buffering, facilitating serverless networking and orchestration. However, the existing sidecar proxy is heavyweight since it is continuously running and incurs excessive overheads, including 2 data copies, 2 context switches, and 2 interrupts (see §5.2) for a **single** request. Moreover, since most serverless frameworks primarily focus on HTTP/REST API [23, 128, 31], additional protocol adaptation is required for specialized use cases, *e.g.,* IoT (Internet-of-Things) with MQTT [62], CoAP [88]. The current design runs protocol adaptation as an individual component, resulting in substantial resource consumption [215]. Having such a heavyweight design may overload serverless environments, especially in resource-limited edge clouds or when handling infrequent workloads (*e.g.,* IoT).

Instead, going a step further and invoking code for execution on a completely event-driven basis without using an individual component can result in substantial resource savings.

Poor dataplane performance for function chaining. Modern cloud-native architectures decompose the monolithic application into multiple loosely-coupled, chained functions with the help of platform-independent communication techniques, *e.g.,* HTTP/REST API, for the sake of flexibility. But, this involves context switching, serialization and deserialization, and data copying overheads. The current design also relies heavily on the kernel protocol stack to handle the routing and forwarding of network packets to and between function pods, all of which impact performance. Although function chaining brings flexibility and resiliency for building complex serverless applications, the decoupled nature of these chains also requires additional components (*e.g.,* a message broker such as Apache Kafka [1], to coordinate communication between functions, and a load balancer like Istio [27]). The resulting complex data pipelines add more network communications for the function chain. All of this contributes to poor dataplane performance (lower throughput, higher latency), potentially compromising service level objectives (SLOs).

In this chapter, we design SPRIGHT [181, 180], a high-performance, event-driven, and responsive serverless cloud framework that utilizes shared-memory processing to achieve high-performance communication within a serverless function chain. We base the design of SPRIGHT on Knative [59], a popular open-source serverless framework. Evaluation results are presented for SPRIGHT and compared with Knative under various realistic serverless workloads in a cloud environment. Our event-driven shared memory processing, includes event-driven proxies (we call them the EPROXY and SPROXY) that significantly reduce

the high resource utilization in the Knative design. This results in much lower latency. SPRIGHT overcomes the challenges of existing serverless computing with the following innovations:

(1) We design the SPRIGHT gateway, a chain-wide component, to facilitate shared memory processing within a serverless function chain. The SPRIGHT gateway consolidates protocol stack processing in the Linux kernel and distributes the payload to the chain.

(2) We design event-driven proxies (*i.e.,* EPROXY and SPROXY) using the eBPF (extended Berkeley Packet Filter [39]), that effectively replace the heavyweight sidecar proxy. We support the functions of metrics collection *etc.*, with much lower CPU consumption. We further utilize the XDP/TC hooks provided by eBPF to improve packet forwarding performance outside the serverless function chain. Compared to the kernel networking stack, the eBPF-based dataplane dramatically lowers latency and CPU consumption.

(3) We implement zero-copy message delivery within a serverless function chain by using event-driven shared memory communication. This avoids the unnecessarily duplicated in-kernel packet processing between functions, achieving high-speed, highly scalable packet forwarding within a serverless function chain. Event-driven shared memory communication helps reduce CPU usage and alleviate penalties when keeping the function chain warm.

(4) SPRIGHT fully exploits the reconfigurability of the eBPF maps to support Direct Function Routing (DFR) within the serverless function chains, which eliminates the dependency on an intermediate routing component (*e.g.,* the message broker in Knative [60]) for function chaining and avoids duplicate processing in the dataplane.

(5) We implement the separation at the function-chain level in SPRIGHT's shared memory

processing by restricting access to a private shared memory to trusted functions of only that chain. The SPROXY further restricts unauthorized access by applying message filtering for inter-function communication.

(6) We optimize protocol adaptation by running it as an event-driven component attached to the SPRIGHT gateway, to avoid unnecessary networking protocol stack processing overhead. This optimization can significantly reduce resource usage.

5.2 Background and Challenges

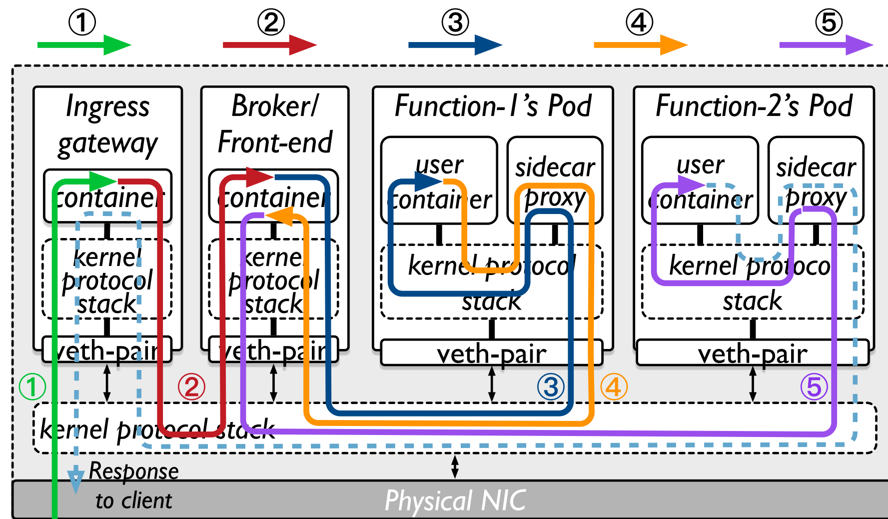


Figure 5.1: Networking processing involved in a typical serverless function chain.

There are a variety of implementations for function chaining since there is no standard for a general solution architecture for serverless applications. The data pipeline patterns for function chaining of different open-source serverless platforms are slightly different, depending on the messaging model applied, *e.g.*, a publish/subscribe model typically

uses a message broker as the intermediate component for coordinating invocations within the function chain, while the request/response model typically employs a front-end proxy to perform invocations within the function chain. We examined the design of several proprietary and open-source serverless platforms [32, 33, 24, 60, 165] and developed a common abstract model of the typical data pipeline pattern they use, as shown in Fig. 5.1.

The data pipeline for function chains uses a message routing as follows: ① Clients send messages (requests) to a message broker/front-end proxy through the ingress gateway of the cluster. ② The messages are queued in the message broker/front-end proxy and registered as an event. ③ The message broker/front-end proxy sends the message to an active pod of the head (first) function in the chain, as defined by the user. ④ The function pod is invoked to process the incoming request. After the first function processes the request, a response is returned and queued in the message broker/front-end proxy, registered as a new event for the next function in the chain. ⑤ The message broker/front-end proxy sends this new event to an active pod for the next function in the chain.

Unfortunately, this data pipeline poses several challenges that are common across the different serverless platforms. The core dataplane components, including the ingress gateway, message broker/front-end proxy, sidecar proxy, *etc.*, are usually implemented as individual, constantly-running, loosely coupled components. In addition, for internal calls within the chain, each involves context switching, serialization/deserialization, and protocol processing.

We quantify the overheads in the representative open-source platform, Knative, through systematic auditing performed with a '1 broker/front-end + 2 functions' chain setup

based on the current design depicted in Fig. 5.1. We assume all evaluated components are deployed on the same node, with the overhead on the external client-side excluded. We use an NGINX [64] server function for this audit. However, our results are generally applicable, as these basic overheads are independent of the function used. We examine the different overheads incurred in the data pipeline processing of one request (from ① to ⑤), including # of copies, # of context switches, *etc.* as listed in Table 5.1. Due to implementation-specific differences, *e.g.,* running multiple threads on the same CPU core, there may inevitably be additional context switches. Our audit aims to quantify the minimum value of each type of overhead. Based on these observations, we list the following key takeaways:

Takeaway#1: Individual, constantly-running heavyweight sidecar. Serverless platforms equip each function pod with an individual, constantly-running sidecar proxy to handle inbound and outbound traffic. The presence of this sidecar proxy introduces a significant amount of overhead. Just going through step ④, the sidecar proxy introduces 2 data copies (50%), 2 context switches (50%), and 2 interrupts (33%). To understand the impact of this overhead on dataplane performance, we evaluate several sidecar proxies, including the Envoy sidecar from Istio [26], Queue proxy from Knative [151], and the OF-watchdog from OpenFaaS [29]. We use these sidecar proxies to work with NGINX [64] as a representative HTTP server function. We also use this NGINX HTTP server function without sidecar proxies as the baseline to quantify the additional overhead introduced by the sidecar proxy. We disable autoscaling and limit ourselves to a single function instance. We use *wrk* [21] as the workload generator and send variable-size HTTP traffic (2% 10KB requests, 98% 100B requests) directly to the function pod on the same node.

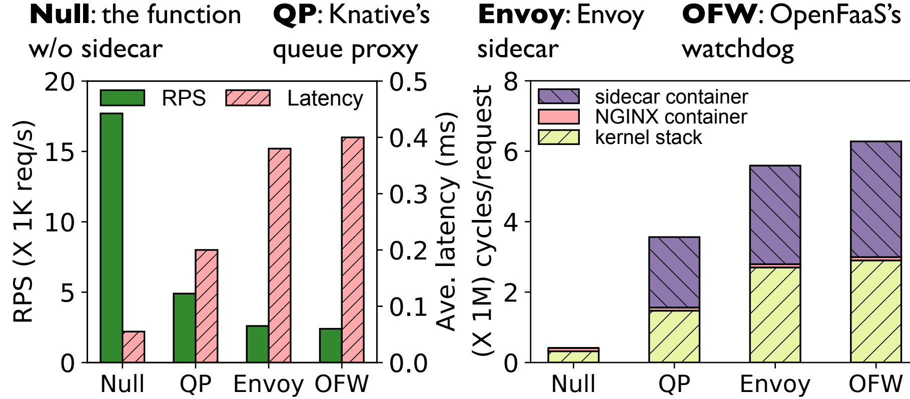


Figure 5.2: Performance and overhead breakdown of different sidecar proxies.

Our experimental results are shown in Fig. 5.2. Equipping a sidecar proxy results in a 3×–7× reduction in throughput, 3×–7× higher latency, and a significant increase (3×–7×) in CPU cycles per request. Even though the overhead varies, it is common across all the evaluated sidecar proxies. Looking deeper at the CPU overhead breakdown, the kernel stack for the sidecar proxy consumes 50% of CPU cycles. This substantial overhead of sidecar proxies undercuts the benefit of serverless computing and calls for a more lightweight serverless capability to provide the same functionality.

Takeaway#2: Excessive data copies, context switches, and interrupts introduced by kernel-based networking. The existing Knative framework uses kernel-based networking to construct the dataplane for a serverless function chain, which inevitably introduces a number of overheads (data copies, context switches, and interrupts) caused by the kernel-userspace boundary crossing. Looking at the network processing from step ① to step ⑤ in Fig. 5.1, each request results in 15 data copies, 15 context switches, and 25 interrupts throughout the entire data pipeline. In particular, most of the overhead (80%) comes from networking within the function chain (from ③ to ⑤). The excessive overhead

adds up as more messages are exchanged between functions, which have to be handled by the kernel. This can greatly impact the dataplane performance of a serverless function chain.

Takeaway#3: Unnecessary serialization & deserialization. HTTP/REST API requires additional serialization and deserialization operations to convert application data to byte streams before being transmitted over the network. These operations incur significant overhead (lowering throughput and adding latency) [218, 187]. Each step in the data pipeline for the function chain (from ③ to ⑤) introduces 2 serialization and 2 deserialization operations. As shown in Table 5.1, current designs further exacerbate this overhead with an excessive number of protocol stack traversals, which we describe next.

Table 5.1: Per request Knative overhead auditing of data pipelines for a '1 broker/front-end + 2 functions' chain.

Data Pipeline No.	*External*			*Within chain*				**Total**
	①	②	*total*	③	④	⑤	*total*	
# of copies	1	2	3	4	4	4	12	15
# of context switches	1	2	3	4	4	4	12	15
# of interrupts	3	4	7	6	6	6	18	25
# of protocol processing tasks	1	2	3	3	3	3	9	12
# of serialization	0	1	1	2	2	2	6	7
# of deserialization	1	1	2	2	2	2	6	8

Takeaway#4: Excessive, duplicate processing. Current approaches for serverless function chains rely on the composition of existing networking components to support asynchronous and reliable message exchange between functions. Traffic within the chain has to go through the message broker/front-end proxy, each time having to cross the kernel-user space boundary. This also leads to duplicate network protocol processing, adding to the

overhead. As seen in Table 5.1, networking *within* the function chain in Knative accounts for 75% of the total protocol processing overhead. Protocol processing tasks, including checksum calculation in software and complex iptables processing,[1] contribute to latency and results in poor scaling (especially as the number of iptables rules increases) [164]. Furthermore, many of the other dataplane overheads (*e.g.,* data copies, context switches, interrupts, and serialization & deserialization) are also amplified, as the chain becomes more complex, resulting in very poor scaling.

Summary: The expected benefit of serverless computing was to overcome the inefficiencies of 'serverful' computing through event-driven execution, which helps use resources strictly on-demand and be proportional to the load. However, the excessive overhead in current serverless frameworks shows that the 'server' is still entrenched in serverless computing. Our auditing shows that the loosely coupled construction of existing components for serverless computing results in substantial unnecessary processing overhead, possibly discouraging the implementation of microservices as function chains. This poor dataplane design and having individual, constantly-running components in the function chain prompt us to create a more streamlined, responsive serverless framework by considering high-performance shared memory processing and lightweight event-driven optimizations to help extract the 'server' out of serverless computing.

[1] In [178] we showed that the overheads for iptables processing in a typical Kubernetes environment (also applicable to Knative) using the Container Network Interface accounts for 60% of the total networking overhead.

5.3 System Design of SPRIGHT

We now provide an overall view of the SPRIGHT architecture, justifying the design of each component and discussing their benefits for serverless environments. We then discuss each part in detail, starting with lightweight event-driven processing (§5.3.2), the shared memory processing for communication within serverless function chains (§5.3.3), and direct function routing (§5.3.4). We also discuss function startup in SPRIGHT (§5.3.5), protocol adaptation (§5.3.6), and security domain design (§5.3.7).

5.3.1 Overview of SPRIGHT

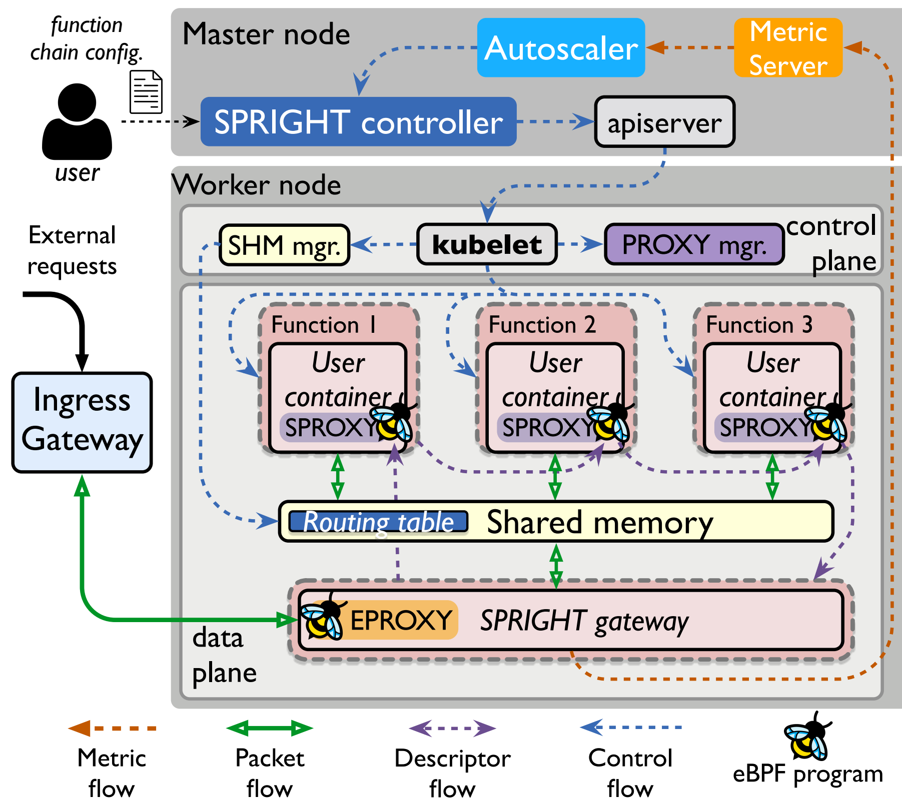


Figure 5.3: The overall architecture of SPRIGHT.

For this work, we start with open-source Knative as the base platform [59]. Using an innovative combination of event-driven processing and shared memory, we achieve high performance while being resource-efficient and providing the flexibility to build microservices using serverless function chaining. Fig. 5.3 shows the overall architecture of SPRIGHT. Importantly, we extensively use eBPF in SPRIGHT for networking and monitoring. eBPF is an in-kernel lightweight virtual machine that can be plugged in/out of the kernel with considerable flexibility, efficiency, and configurability [39]. The execution of eBPF programs is triggered only whenever a new event arrives, thus working naturally with the event-driven serverless environment. Using eBPF, various event-driven programs can be attached to kernel hook points (*e.g.,* the network or socket interface). This enables high-speed packet processing [125] and low-overhead metric collection [149]. eBPF achieves its configurability through eBPF maps – a configurable data structure shared between the kernel and userspace. With eBPF maps, a more flexible dataplane can be implemented with customized routing. The good features of eBPF help us provide functionality with strictly load-proportional resource usage, a highly desirable toolbox for serverless environments.

SPRIGHT's dataplane: SPRIGHT improves the dataplane of serverless computing by leveraging eBPF-based event-driven processing and shared memory communication, which helps us avoid the use of individual, constantly-running sidecars (Takeaway#1 in §5.2) and reduces a number of data movement related overheads within the function chain (Takeaway#2 and #3 in §5.2). SPRIGHT uses Direct Function Routing (DFR) to forward requests directly from one function to the next. This eliminates the need for an intermediate routing component and avoids unnecessary, duplicate processing overheads (Takeaway#4 in

§5.2). These dataplane optimizations make the request handling in SPRIGHT strictly load-proportional and achieve superior performance compared to existing serverless platforms (see evaluation in §5.4).

- **Event-driven processing:** We design lightweight, event-driven proxies (EPROXY and SPROXY in Fig. 5.3) that use eBPF to construct the service mesh instead of a continuously-running sidecar proxy associated with each function instance, as is used by Knative. Thus, we reduce a significant amount of the processing overhead (§5.3.2).

  To accelerate the data path to/from the function chain and the ingress gateway which is outside the function chain, we utilize XDP/TC hooks [188] in eBPF (§5.3.2). An XDP/TC hook processes packets at the early stage of the kernel receive (RX) path before packets enter into the kernel iptables [83], resulting in substantial dataplane performance improvement without the resource consumption of a dedicated sidecar proxy that uses the kernel protocol stack. In addition, protocol adaptation is often required to interface between application layer protocols, such as MQTT and CoAP for IoT, to the HTTP/REST API supported by serverless frameworks (addressed in §5.3.6).

- **Shared memory communication:** For inter-function communication within the chain, SPRIGHT takes advantage of shared memory that avoids a number of overhead associated with data movement, including protocol processing, serialization/deserialization, memory-memory copies, *etc.* For every incoming request from external clients, a SPRIGHT gateway performs the one-time, consolidated protocol processing for the function chain (§5.3.3). SPRIGHT considers event-driven `SKMSG`, which is a socket-

related eBPF program type [188], to construct the zero-copy I/O (*i.e.,* descriptor delivery) between functions (see §5.3.3).

- **Direct function routing (DFR):** To eliminate the impact of having an intermediate routing component (message broker) within the function chain, we design Direct Function Routing (DFR). DFR leverages the configurability provided by eBPF maps and allows for the dynamic update of routing rules while exploiting shared memory to pass data directly between the functions within the chain (§5.3.4).

Although these dataplane optimizations are built around the Knative, our concepts and methodology can also be broadly applied to other serverless platforms. In addition to dataplane optimizations, SPRIGHT incorporates security domains to restrict unauthorized access between different chains, by creating a private shared memory pool for each chain and applying message filtering for inter-function communication [181].

SPRIGHT's control plane: We introduce a SPRIGHT controller (Fig. 5.3) to coordinate the control plane for SPRIGHT function chains. The SPRIGHT controller runs as a cluster-wide control plane component in the master node, working with serverless orchestration engines, *e.g.,* Knative and Mu [166], and their associated control plane components (*e.g.,* autoscaler, placement engine) to determine the scale and placement of the function chain at the appropriate worker node.

SPRIGHT adopts Kubernetes to manage the lifecycle of function pods (*e.g.,* pod creation, termination). It cooperates with the *kubelet*, which is a key pod management process in the Kubernetes control plane that runs on each worker node to manage the lifecycle of the pods. Given a function chain creation request from the SPRIGHT controller, the

kubelet starts up functions in the chain based on the user configuration, working in conjunction with the shared memory manager ('SHM mgr.' in Fig. 5.3) and PROXY manager to set up the dataplane for the function chain. This process includes initializing the shared memory pool, creating the SPRIGHT gateway, and attaching eBPF programs to functions (details in §5.3.2). Each worker node has a shared memory manager and a PROXY manager in the control plane, both running as separate Kubernetes pods. To route external requests to the SPRIGHT gateway of each function chain, we use a cluster-wide Ingress Gateway to distribute the traffic.

5.3.2 Event-driven processing

eBPF-based event-driven proxy (EPROXY/SPROXY) The sidecar proxy in a serverless environment, *e.g.,* the queue proxy in Knative, runs as an additional container in a function pod distinct from the user container. It buffers incoming requests before forwarding them to the user container, to help handle traffic bursts and maintain throughput. The sidecar proxy is also responsible for collecting metrics for the pod (*e.g.,* request rate, concurrency level, response time) and exposing them to a metrics server to facilitate control plane decision-making, *e.g.,* autoscaling. However, this design has several drawbacks as we described earlier. We overcome them with our lightweight, event-driven eBPF-based EPROXY & SPROXY, which replace the sidecar proxy.

The EPROXY is composed of a set of eBPF programs executed at the veth of the SPRIGHT gateway (Fig. 5.4), using XDP and TC hooks [188]. SPRIGHT uses EPROXY to perform L3 metrics collection and dataplane acceleration (§5.3.2). The SPROXY runs

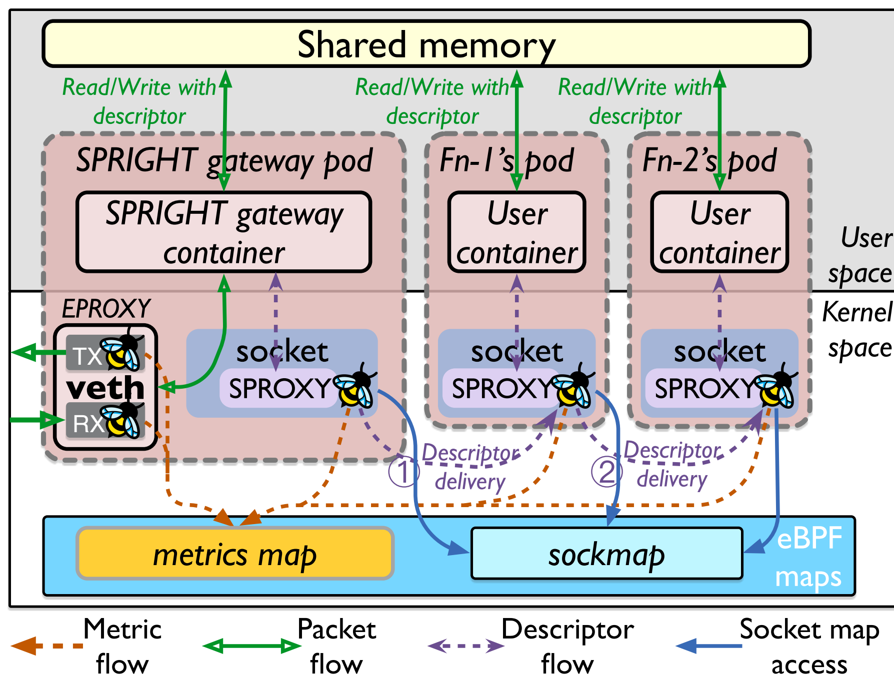


Figure 5.4: Event-driven EPROXY & SPROXY, shared memory, and DFR within a chain: (①) the SPRIGHT gateway invokes the head function of chain; (②) the head function calls the next function bypassing the SPRIGHT gateway.

as a set of socket-related eBPF programs (SK_MSG [188]) at the socket interface of the SPRIGHT gateway/function pods (Fig. 5.4). The 'SK_MSG' program supports modification of messages that pass through the attached socket as well as message redirection between sockets (with the help of eBPF's sockmap [188] to provide routing rules), which is an ideal capability to exchange small messages such as packet descriptors in supporting shared memory communication (§5.3.3). However, the 'SK_MSG' program only works on the TX path of the socket [188]. We use SPROXY for L7 metrics collection, packet descriptor exchange (§5.3.3), routing (§5.3.4), and security [181].

The goal of the event-driven proxy is to achieve functionality comparable to that of the sidecar proxy but with lower overhead. Since the event-driven proxy is only triggered when there are incoming requests, there is no CPU overhead when idle. Although EPROXY and SPROXY work in the kernel, they are created by the cloud service provider rather than the user, which does not affect the isolation of the user function. This is similar to how serverless platforms attach a sidecar to a user function. We do not need the queueing capability in the event-driven proxy as the shared memory within the function chain already provides that queueing. Thus, SPRIGHT still provides the same functionality to improve concurrency and handle traffic bursts as a sidecar proxy. But, eliminating the additional queuing stage helps reduce request delays.

Metrics collection: To collect the required metrics for SPRIGHT's control plane, we attach eBPF-based monitor programs to the EPROXY and SPROXY. The EPROXY is used to collect L3 metrics, *e.g.,* packet rate, bytes received, while the SPROXY collects L7 metrics, *e.g.,* request rate. We assign a 'metrics map' in the eBPF maps that serves as a

local metrics storage on each node. When a new message occurs, the monitor programs are triggered to collect and update the metrics to the metrics map. The SPRIGHT gateway has a built-in metrics agent responsible for reading the metrics map periodically and providing the latest metrics to the metrics server. We further extend the SPRIGHT gateway with internal event-driven metrics collection capabilities as an enhancement of EPROXY to provide function-chain-level metrics such as the request rate and execution time on a chain basis.

Function health checks: Kubernetes natively supports function pod health checks via the *kubelet*. Working in conjunction with the *kubelet*, SPRIGHT can enable TCP or HTTP probes in functions for health checks. Enabling the TCP or HTTP probes requires a minimal change of opening an additional socket or HTTP server in the function to listen to health check requests from the control plane. Thus, we can dispense with Knative's sidecar proxy doing a health check to check on function pods, using TCP or HTTP probing.

Initialization of EPROXY & SPROXY: We dedicate a PROXY manager (Fig. 5.3) on each worker node for attaching EPROXY/SPROXY to the SPRIGHT gateway and/or function pods. The PROXY manager is created during the startup of the worker node's control plane, during which it loads the EPROXY/SPROXY eBPF programs into the kernel (via the *bpf()* syscall) and creates required eBPF maps, including the metrics map and socket map (*sockmap*). The `SKMSG` program is then attached to the sockmap. The sockmap automatically attaches the `SKMSG` program to the function pod's socket interface, once a function's socket interface is registered into it by the sockmap writer in the PROXY manager. The initialization procedure of the EPROXY starts as soon as the SPRIGHT

gateway is ready (*i.e.,* when SPRIGHT gateway passes the health check from *kubelet*). *kubelet* instructs the PROXY manager to attach the EPROXY programs to the XDP and TC hooks at the veth of SPRIGHT gateway.

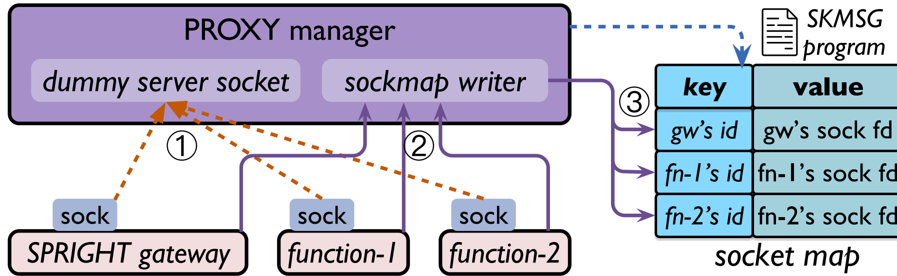


Figure 5.5: Initialization of SPROXY: `SKMSG` program, sockmap.

Both the SPRIGHT gateway and functions follow the same procedure to attach to the SPROXY. We use a function as an example to explain the initialization procedure of SPROXY, as depicted in Fig. 5.5. During startup, the function creates a socket interface for attaching to the SPROXY. The function initiates a connection on its socket to the dummy socket in the PROXY manager. The dummy socket stays active to maintain the connection with function's socket (①). After the connection is established, the function sends the socket's file descriptor and the function ID to the PROXY manager (②). To attach the `SKMSG` program to function's socket, the PROXY manager uses its sockmap writer to update the function ID (key) and the socket's file descriptor (value) to the sockmap (③), using '*bpf_map_update_elem()*' helper. The `SKMSG` program is then automatically attached to the socket by the kernel.

eBPF-based dataplane acceleration for external communication We exploit EPROXY's XDP/TC hooks to accelerate the communication by the function chain in SPRIGHT to external components (*e.g.,* cluster-wide Ingress Gateway). We develop an eBPF forwarding program (in EPROXY) and attach it to the XDP/TC hook that is positioned on the RX path of the network interface, including the host-side veth of the pod (*i.e., veth-host*[2]) and the physical NIC, as shown in Fig. 5.6. eBPF offers packet redirect features (*i.e.,* 'XDP_REDIRECT' and 'TC_ACT_REDIRECT') that support passing raw frames between the virtual network interfaces, or to and from the physical NIC without going through the kernel protocol stack [55]. This helps save CPU cycles consumed by iptables. The eBPF forwarding program has two functions: 1) Look up the kernel FIB (Forwarding Information Base) table to find the destination network interface based on the FIB parameters of the received packet (using *bpf_fib_lookup()* helper), including the IP 5-tuple, index of source interface, *etc.* 2) Forward the raw packet frame to the target (*veth-host* or NIC) interface via 'XDP_REDIRECT' or 'TC_ACT_REDIRECT'. The communication could be either in the same node or across different nodes, supported by an eBPF-based dataplane via the eBPF forwarding program. An XDP program at the physical NIC processes all inbound packets received by the NIC. It redirects the packet to the *veth-host* of the destination function pod after a routing table lookup (① in Fig. 5.6). The TC program at the *veth-host* handles the outbound packet from the function pod. Depending on the packet's destination, the TC program may take different routes. If the destination of the packet is to another function pod (*e.g.,* traffic between ingress gateway pod and SPRIGHT gateway pod) on the same

[2]A function pod is connected to the host through a pair of veths, *i.e.,* the host-side veth and pod-side veth.

node, the TC program directly passes the packet to the *veth-host* of the destination function pod via 'TC_ACT_REDIRECT' (② in Fig. 5.6). If the destination function pod is on another node, the TC program redirects the packet to the NIC (③ in Fig. 5.6).

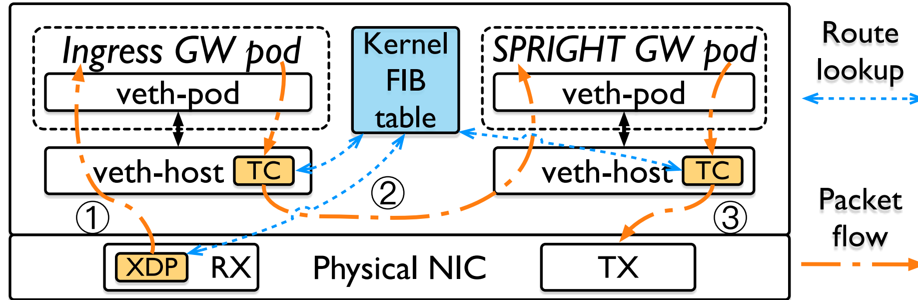


Figure 5.6: Dataplane acceleration using eBPF XDP/TC hooks

Improvement with dataplane acceleration based on EPROXY: To estimate the benefit of eBPF's XDP/TC features, we evaluate the networking performance of SPRIGHT when the XDP/TC acceleration is enabled in EPROXY. We use Apache Benchmark [46] to simulate the traffic to/from the cluster-wide ingress gateway running on a different node than the SPRIGHT gateway. We further break down the CPU cycles expended for the kernel stack processing, to accurately quantify the CPU cycles saved by the XDP/TC acceleration.

Fig. 5.7 (Left) compares the RPS and response latency performance of SPRIGHT when the XDP/TC acceleration is enabled or disabled. With a concurrency of 32, SPRIGHT with XDP/TC acceleration has a 1.3× improvement in RPS compared to SPRIGHT without XDP/TC acceleration. Since XDP/TC acceleration transfers raw packets between network devices (*i.e.,* veth and NIC), the overhead spent in kernel iptables can be avoided, which in turn improves throughput. The response latency of SPRIGHT with XDP/TC acceleration is 19μs with a concurrency of 32, compared to 24μs for SPRIGHT without XDP/TC

acceleration. The RPS and response latency improvements remain even as the concurrency increases, allowing SPRIGHT to maintain a peak RPS of 53K when XDP/TC acceleration is enabled.

We then break down the CPU cycles spent on processing a request (at a concurrency of 32) based on where the cycles are expended, including in the host's kernel stack and the pod's network stack, as shown in Fig. 5.7 (Right). The client-side overhead is excluded. For SPRIGHT without XDP/TC acceleration, about 15.2K CPU cycles are spent on the host's kernel networking layer for iptables processing. Whereas with XDP/TC acceleration, since iptables processing in the host's kernel stack is skipped, only 2.1K CPU cycles are consumed by the host's kernel networking layer, resulting in 86% of CPU cycles being saved for each request. This clearly demonstrates the benefits of eBPF's XDP/TC acceleration. Bypassing the host's kernel networking stack and associated iptables processing can save considerable CPU usage and thus benefit SPRIGHT's dataplane performance for communication outside the function chain. This option, however, means the loss of full-featured iptables network policy support, which may not be required for certain cases (*e.g.,* when users require higher dataplane performance, with the infrastructure provider having only a subset of the kernel iptables functionality [73]).

5.3.3 Shared memory communication within function chains

To support shared memory communication within a serverless function chain, three key building blocks are required: (1) **Protocol processing.** The incoming message to a SPRIGHT function chain requires protocol processing before being moved to shared memory

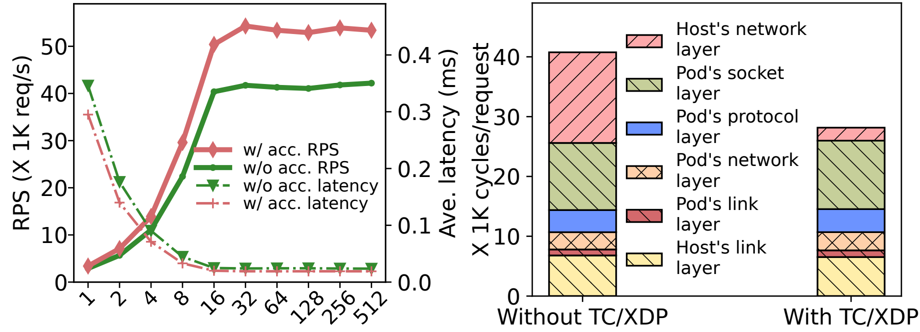


Figure 5.7: (Left) Performance impact of TC/XDP redirect: RPS and latency; (Right) CPU overhead breakdown of receiver side kernel stacks: with TC/XDP acceleration (w/ acc.) & without TC/XDP acceleration (w/o acc.).

for communication within the function chain. Similar protocol processing to construct outgoing messages is needed. (2) **Shared memory pool.** A private shared memory pool that is initialized for the function chain and is attached to functions during their startup is needed. The message payload is kept in shared memory without being moved between functions. (3) **Zero-copy I/O within the function chain.** To enable zero-copy data movement between functions, shared memory processing relies on packet descriptors to pass the location of data in the shared memory pool, which is then accessed by the function.

Consolidated protocol processing To flexibly manage traffic in and out of the function chain and avoid duplicate protocol processing within the chain, we create a SPRIGHT gateway. It acts as a reverse proxy for the function chain to consolidate the protocol processing. The SPRIGHT gateway relies on the kernel protocol stack for protocol processing and extracts the application data (*i.e.,* Layer 7 payload). It intercepts incoming requests to the function chain and copies the payload into a shared memory region. This enables

zero-copy processing within the chain, avoids unnecessary serialization/deserialization and protocol stack processing. The SPRIGHT gateway invokes the function chain for requests, processes the results, and constructs the HTTP response to external clients. SPRIGHT assumes that functions in the same chain run within the same node, to derive the benefits of sharing the memory between functions.

We dedicate a SPRIGHT gateway for each function chain to support the security domain isolation between different chains. To mitigate the concern of overhead when there are many chains in the cluster, we emphasize that the SPRIGHT gateway is a lightweight component with a relatively small memory footprint (27KB compared to each (even simple) Golang-based function that is more than 2MB). In addition, since the SPRIGHT gateway processes requests based on kernel interrupts, its event-driven nature results in the CPU usage being largely load-dependent.

Shared memory pool SPRIGHT allocates a private shared memory pool with Linux HugePages for each serverless function chain. Using HugePages can reduce the access overhead of in-memory pages, thus improving the performance of serverless functions when accessing data in the shared memory pool. In addition, the shared memory pool within the function chain supports queueing to help sustain traffic bursts.

SPRIGHT takes advantage of DPDK's multi-process support [53] to create shared memory pools for function chains. At the startup of a SPRIGHT function chain, a DPDK primary process is spun up in the shared memory manager pod. The DPDK primary process has the privileged permission to initialize the shared memory pool, using *rte_mempool_create()* API. Each DPDK primary process owns a unique shared data file prefix [53] – a

multiprocessing-related option in DPDK. We further extend its use to isolate different memory pools [181]. By specifying the correct prefix, the gateway and functions in SPRIGHT, which run as DPDK secondary processes, can attach to the memory pool (use *rte_memzone_lookup()* API) created by the chain's specific DPDK primary process in the shared memory manager pod.

Note that DPDK's multi-process shared memory is *independent* of other DPDK libs/devices such as DPDK RTE RING and Poll Mode Driver. This gives SPRIGHT the freedom to choose different implementations of zero-copy I/O to support shared memory communication within the function chain.

Event-driven zero-copy I/O within the function chain SPRIGHT extends the use of SPROXY (Fig. 5.4) to implement the *event-driven* zero-copy I/O for shared memory communication within the function chain. The `SKMSG` program in SPROXY works with eBPF's sockmap to enable message redirection between the socket interfaces of function pods by communicating a packet descriptor from one function to the next in the chain. The packet descriptor used in SPRIGHT is a small 16-byte message that incurs negligible overhead. A packet descriptor contains two fields: the instance ID of the next function and a pointer to the data in shared memory. Once the SPROXY receives a packet descriptor, it extracts the instance ID of the next function, which is then used to query the eBPF's sockmap to retrieve the target socket interface information (*i.e.,* the file descriptor). For the description of the zero-copy based message flow in SPRIGHT, refer to [181].

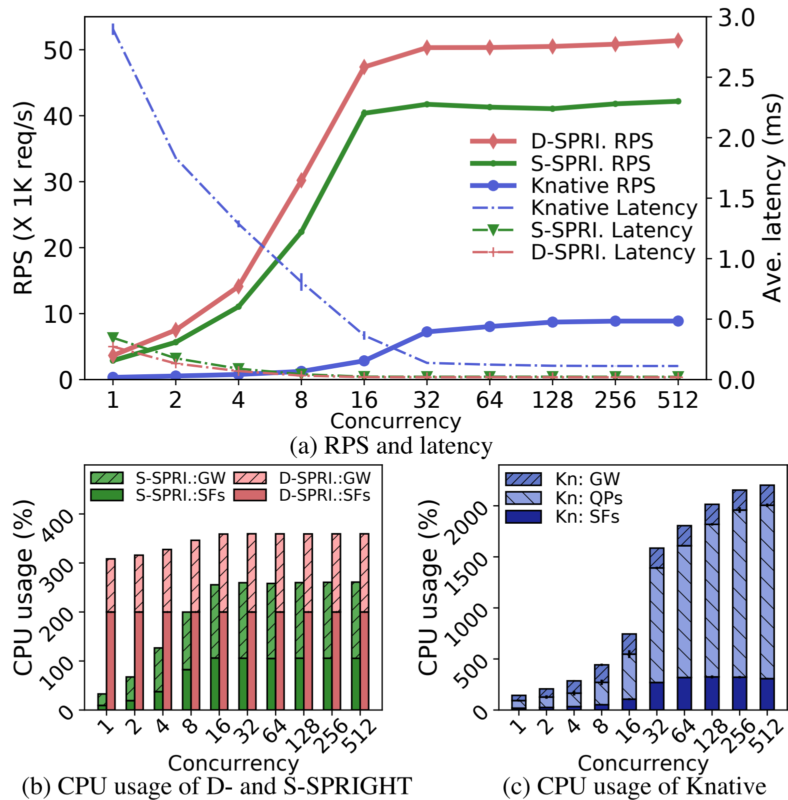
The packet descriptor redirection performed by the SPROXY bypasses any kernel protocol stack processing (which is unnecessary here), incurring minimal overhead.

SPROXY operates in a purely event-driven manner, avoiding the need to busy-poll descriptors and saving CPU resources. Thus, the communication overhead is entirely load-dependent.

Another implementation option is using polling-based zero-copy I/O, as used by DPDK, which uses polling-based RTE RING [69] to pass packet descriptors. DPDK's RTE RING is implemented as a userspace shared memory queue that offers a low-latency IPC channel between independent processes (*i.e.,* function pods) because it entirely eliminates any kernel-userspace interaction (*e.g.,* context switches, interrupts) and operates at memory speeds, ensuring higher performance. DPDK's RTE RING has been extensively used to build high-performance dataplanes for cloud services [226]. However, using DPDK's RTE RING as the inter-function IPC channel requires expensive busy polling that continuously consumes CPU cycles, independent of traffic intensity.

Event-driven vs. polling-based shared memory processing To identify the most appropriate zero-copy I/O for shared memory processing in the context of serverless computing, we compare SPRIGHT's event-driven shared memory processing based on SPROXY (hereafter referred to as S-SPRIGHT) with polling-based shared memory processing based on DPDK (hereafter referred to as D-SPRIGHT), with a function chain containing 2 function pods. We use Apache Benchmark [46] on a second node as the workload generator. We additionally set up a function chain with the base Knative environment and use NGINX as the front-end proxy to coordinate the communication within the chain. Both the SPRIGHT gateway and NGINX proxy are configured with two dedicated cores for a fair comparison. Note: We collect the results from 10 repetitions.

As shown in Fig. 5.8, with low concurrency, *e.g.,* at 32, S-SPRIGHT (0.024ms) shows a slightly higher average response delay compared to D-SPRIGHT (0.02ms), but still has a much lower (almost 6×) response latency compared to Knative (0.138ms). In terms of RPS, both D-SPRIGHT (50.3K) and S-SPRIGHT (41.7K) are substantially higher than Knative (7.2K), with a significant 5.7× improvement.



(a) RPS and latency

(b) CPU usage of D- and S-SPRIGHT

(c) CPU usage of Knative

Figure 5.8: Comparison between polling-based (D-SPRI.) and event-driven (S-SPRI.) shared memory processing with 1 gateway pod and 2 function pods. Kn: Knative; QPs: Sidecars; SFs: serverless functions; GW: gateway. All results show the 99% confidence interval.

As S-SPRIGHT relies on the in-kernel eBPF program (*i.e.,* SPROXY) to deliver packet descriptors, it incurs the overheads for context switching, contributing to the extra latency. However, the SPROXY processing latency is masked when the concurrency increases (≥ 32), because the context switching latency overlaps with the other processing. Throughput increases rapidly, up to 5× that of Knative. Although S-SPRIGHT has a 1.2× lower peak throughput than D-SPRIGHT, S-SPRIGHT has a substantially lower CPU usage, because it is purely event-driven. Both of those approaches have a much lower overhead compared to Knative. With a concurrency of 1, S-SPRIGHT consumes 32% CPU, which is 9.6× and 4.5× less than D-SPRIGHT (308%, or more than 3 CPU cores fully used) and Knative (143%), respectively. When the concurrency increases to 32, S-SPRIGHT consumes 259% CPU, which is still less than D-SPRIGHT (359%). Comparatively, the CPU usage of base Knative increases to a shocking 1585% (more than 15 CPU cores used) at a concurrency of 32 (see Fig. 5.8 (c)). The sidecar proxy consumes 70% of Knative's CPU. Even with increasing concurrency (≥ 32), S-SPRIGHT has a consistent and steady saving in CPU compared to the others. Individual, constantly-running components (sidecar proxy with Knative or DPDK's poll mode using up CPUs) have excessive overhead. More importantly, S-SPRIGHT consumes negligible CPU resources when there is no traffic. We observed that S-SPRIGHT's gateway and function pods that are event-driven consume *zero* CPU when there is no traffic, making it possible to keep a function pod 'warm' to overcome the 'cold start' delay (§5.4.2). Thus, event-driven shared memory processing is ideal for serverless computing, especially for function chains.

5.3.4 Direct Function Routing within a function chain

To optimize the invocations within a function chain, we use Direct Function Routing (DFR), which enables the upstream function in the chain to directly invoke/communicate with the downstream function. As shown in Fig. 5.4, the SPRIGHT gateway only invokes the head function in the chain once (① in Fig. 5.4). When the first function completes the request message processing (② in Fig. 5.4), it directly calls the next function without going through the SPRIGHT gateway. The rest of the function invocations in the chain also bypass the SPRIGHT gateway, thus significantly reducing the invocation latency (and overhead) for the function chain. To support DFR, SPRIGHT adopts a two-step routing mechanism. It uses a chain-specific, userspace routing table, and an in-kernel sockmap. The userspace routing table helps determine the ID of next function while the in-kernel sockmap uses that function ID to find its corresponding socket file descriptor, which is then used by the SPROXY to perform the actual packet descriptor delivery between the sockets of the source and destination function.

We use the SPRIGHT controller (Fig. 5.3) to manage DFR within the function chain. The SPRIGHT controller configures the routing table based on the user-defined sequence for the function chain. We keep the routing table in shared memory to reduce access latency. To support multiple downstream functions, we use a 'topic' (extracted from the message payload) based publish/subscribe messaging model, and dynamically route requests using the routing table. The message topic and the ID of the current function serve as the key to looking up the ID of the next-hop function in the routing table. For details of load balancing with a function chain, refer to [181].

5.3.5 Function startup in SPRIGHT

In Knative, the startup of a function pod consists of several key steps: control plane activity (*e.g.,* pod placement), container runtime initialization (*e.g.,* container image extraction, namespace creation, Cgroups configuration, file system mounting, etc), and dataplane setup (*e.g.,* route setup, veth devices creation, etc). The startup process of a SPRIGHT function pod shares several common steps with a Knative function pod creation in terms of control plane activity and container runtime initialization. However, SPRIGHT differs from Knative in setting up the dataplane because SPRIGHT uses shared memory communication. More importantly, SPRIGHT uses eBPF-based event-driven proxies. Knative, on the other hand, uses the sidecar proxy as an individual container, thus incurring additional startup latency to initialize the sidecar container. The dataplane setup of a Knative function pod is nested within the container runtime initialization and is completed by the Container Network Interface (CNI) plugin [178]. During the dataplane setup of a Knative function pod, the CNI plugin creates a veth-pair (a pod-side veth and a host-side veth) to connect the function pod to the host's network namespace, facilitating inter-pod connectivity. An IP address is assigned to the function pod and the route is configured in the host's iptables to finalize the dataplane setup [178].

By using shared memory and the SPROXY for communication within the function chain, SPRIGHT avoids relying on the CNI plugin to set up the dataplane of the function pod. The dataplane setup of a SPRIGHT function pod involves the initialization of the SPROXY (§5.3.2) and attachment to the shared memory pool (§5.3.3). We compare the startup overhead of SPRIGHT function pods and Knative function pods in §5.4.3. *Note:*

The use of shared memory in SPRIGHT requires initialization of the shared memory pool, which may incur considerable latency. To avoid this shared memory creation latency during dataplane setup for SPRIGHT functions, SPRIGHT pre-allocates a number of shared memory objects in the shared memory pool. This is done while initializing the DPDK primary process in the shared memory manager pod.

When starting up a function chain in SPRIGHT, a SPRIGHT gateway pod is created and dedicated to the function chain to perform protocol processing and move requests to/from shared memory. The startup of the SPRIGHT gateway pod involves the initialization of SPROXY and attachment to the shared memory pool. Additional dataplane setup (*e.g.,* veth-pair creation, route setup, etc) is performed by the CNI plugin to connect the SPRIGHT gateway pod with the kernel protocol stack, as the SPRIGHT gateway interacts with the kernel protocol stack to perform protocol processing and to attach the EPROXY to the veth device. The initialization of the SPRIGHT gateway pod as well as the startup of functions in the same chain can be performed in parallel to amortize the startup penalty, as we discuss in §5.4.3.

5.3.6 Event-driven protocol adaptation

Event-driven processing can help tremendously in interfacing serverless frameworks, which have an HTTP/REST API, with a variety of application-specific protocols (*e.g.,* for IoT with MQTT [62], CoAP [88]). Current designs use a separate protocol adapter (*e.g.,* Kamelet in Apache Camel-K [20]) for translation between these protocols. However, since SPRIGHT's shared memory processing directly works on payloads independent of the application layer protocols, the protocol adapter can ideally run as an internal event-driven

component that is part of the SPRIGHT gateway. This achieves a much more streamlined protocol adapter design, using resources strictly on demand. Please refer to [181] for the details of the event-driven protocol adaptation design.

5.3.7 Security domains in SPRIGHT

SPRIGHT recognizes the need for isolation between serverless functions in a shared cloud environment, especially with the use of shared memory processing. It is necessary to restrict access of a shared memory pool to only trusted functions. The trust model in SPRIGHT assumes that the functions within a chain trust each other, but the functions in different chains may not. To limit unauthorized access across function chains, SPRIGHT provides two abstractions to construct a security domain for each function chain: 1) a private shared memory pool for each chain; 2) inter-function packet descriptor filtering with the SPROXY. Details of SPRIGHT's security domain design are in [181].

The current SPRIGHT security domain design requires a dedicated SPRIGHT gateway for each function chain. Although the SPRIGHT gateway is designed to be event-driven and consumes no CPU cycles when idle, the SPRIGHT gateway is assigned dedicated CPU cores to minimize interference. This means that the CPU cores are used exclusively by the assigned SPRIGHT gateway instances, potentially leading to wasted CPU resources, especially when there is an imbalanced load between different SPRIGHT gateway instances. Using a common file prefix for all SPRIGHT function chains on the same node would allow a common SPRIGHT gateway to be shared across all function chains, thus helping to multiplex a single gateway instance and improve CPU utilization. However, using a common file prefix for all SPRIGHT function chains will weaken the isolation between

different shared memory pools. We strike the difficult balance of favoring isolation and thus use a dedicated SPRIGHT gateway instance.

5.3.8 Discussion

Overhead auditing (contd.) - SPRIGHT: We now perform an audit of overhead for SPRIGHT, following the same methodology used before in §5.2, and compare it against the base design depicted in Fig. 5.1. As can be seen in Table 5.2, SPRIGHT significantly reduces overheads for processing within the function chain. With shared memory processing, SPRIGHT achieves **0** data copies, **0** additional protocol processing, and **no** serialization/deserialization overheads within the chain. Although the use of SPROXY generates context switches and interrupts, which do add latency for processing, the total number of context switches and interrupts for SPRIGHT is still much less than that of the base Knative design (repeated in the last column of Table 5.2). In addition, the results in Fig. 5.8 show that the context switches and interrupts introduced by SPROXY have a limited impact on the performance with concurrent processing of just a few sessions. The event-based shared memory processing substantially reduces resource usage, more than compensating for any of the added context switches and interrupts.

Deployment Constraints: In SPRIGHT, functions in the same chain need to be placed on the same node to allow shared memory communication. This requires the placement engine to deploy functions on the basis of a chain. In addition, scaling SPRIGHT across multiple nodes requires all the functions of a chain to be deployed on each node. This may lead to more resource fragmentation compared to deployment of each function.

Table 5.2: Per request data pipeline overhead for SPRIGHT

Data Pipeline No.	*External*			*Within chain*			Total	Total of Kn
	①	②	*total*	③	④	*total*		
# of copies	1	2	3	0	0	0	3	15
# of context switches	1	2	3	2	2	4	7	15
# of interrupts	3	4	7	2	2	4	11	25
# of proto. processing tasks	1	2	3	0	0	0	3	12
# of serialization	0	1	1	0	0	0	1	7
# of deserialization	1	1	2	0	0	0	2	8

Note: 1. We audit a '1 broker/front-end + 2 functions' chain and exclude client overhead; 2. as SPRIGHT uses DFR, hence no route ④ and ⑤ in Fig. 5.1. ④ here means direct route from function-1's pod to function-2's pod.

Application Porting Requirements: Porting an HTTP/REST-based application to SPRIGHT requires replacing HTTP/REST-based I/O with SPRIGHT's event-driven shared memory I/O. However, SPRIGHT does not support synchronous calls between functions, *e.g.,* where the client sends a request to the server and waits for a response before moving on to the next step. SPRIGHT's existing programming model assumes that a function's code runs to completion after invocation, is purely event-driven, and inherently supports asynchronous calls between functions. In SPRIGHT, a synchronous call needs to be broken down into multi-step asynchronous calls. For instance, a synchronous "request-response" transaction needs to be treated as two separate asynchronous calls: (1) The calling function sends a request (associated with a caller ID) to the serving function. The calling function can continue processing other requests without pausing and waiting for the response. (2) The serving function generates a response and sends it back to the calling function (based on the caller ID in the request). The calling function receives the response and then continues to process the transaction. We ported the online boutique application [65] to SPRIGHT

and evaluate it in §5.4.2. The source code is available in [34]. Currently, SPRIGHT only supports C-based implementations of user functions. We expect to extend SPRIGHT's language support as part of our ongoing development of SPRIGHT.

5.4 Evaluation & Analysis

5.4.1 Experiment Setup

To examine the improvement of SPRIGHT and its components, we consider several typical serverless scenarios, including (1) a popular online shopping boutique, (2) An IoT environment of motion detectors, and (3) a more complex processing of image detection & charging for an automated parking garage. For each scenario, we set up a function chain to execute the serverless application (Fig. 5.9). The details of the setup for each scenario are as follows:

1. Online Boutique is an open-source representative implementation of a microservice-based online store application [65]. It has 10 different functions communicating with each other using gRPC. We ported these functions to SPRIGHT (in C) and Knative (using Go language) based on the implementation provided in [65]. Functions ported to SPRIGHT use shared memory, while functions in Knative continue to use gRPC, for inter-function communication. We use Locust [28] as the load generator and use the default workload provided in [65] to generate a realistic web-based shopping application's request pattern. The default workload utilizes a total of 6 different sequences of function chains (see [34]). We compare four alternatives to run the online boutique application, including gRPC, Knative, S-SPRIGHT, and D-SPRIGHT. In the "gRPC" mode ('server-full' approach), the function

runs as a Kubernetes pod without a sidecar and uses the built-in gRPC server for functions to talk to each other directly without involving a broker/front-end.[3] In Knative mode, we use the Istio ingress gateway to mediate the communication between functions. We disable the activator [61] (a cluster-wide queuing component in Knative) to avoid additional queuing delays.

2. IoT - Indoor motion detection for automated lighting requires tracking a sequence of events utilizing multiple sensors. The simple function chain contains 2 functions (Fig. 5.9 (b)). Motion sensors going 'on' triggers an actuator function to turn on the light. The light may be automatically turned off after a period of no activity. We consider the MERL motion detector dataset [219]. We use a traffic generator developed in Python to send motion events based on the timestamps in the dataset. The CPU service time of the sensor function and actuator function are both set at 1ms. For the base Knative setup, we use NGINX to coordinate the communication within the function chain.

3. Parking - image detection & charging takes snapshots of each parking spot as input for visual occupancy (of parking spots) detection in parking lots. It detects the vehicle's license plate and determines whether the plate metadata is stored in the database through a plate search function. If it is not stored, a 'persist-metadata' function is invoked to store the plate metadata in the database. Finally, it charges parking fees based on the license plate's metadata. We consider the *CNRPark+EXT* image dataset collected from a parking lot with 164 parking spaces [78]. We use the same load generator used for IoT

[3]The "frontend service" (Fig. 5.9 (a)) in the online boutique runs as a user function, which is distinct from the general broker/front-end. The latter is a system component that used to mediate the communication between functions (*e.g.,* the Istio ingress gateway in Knative mode).

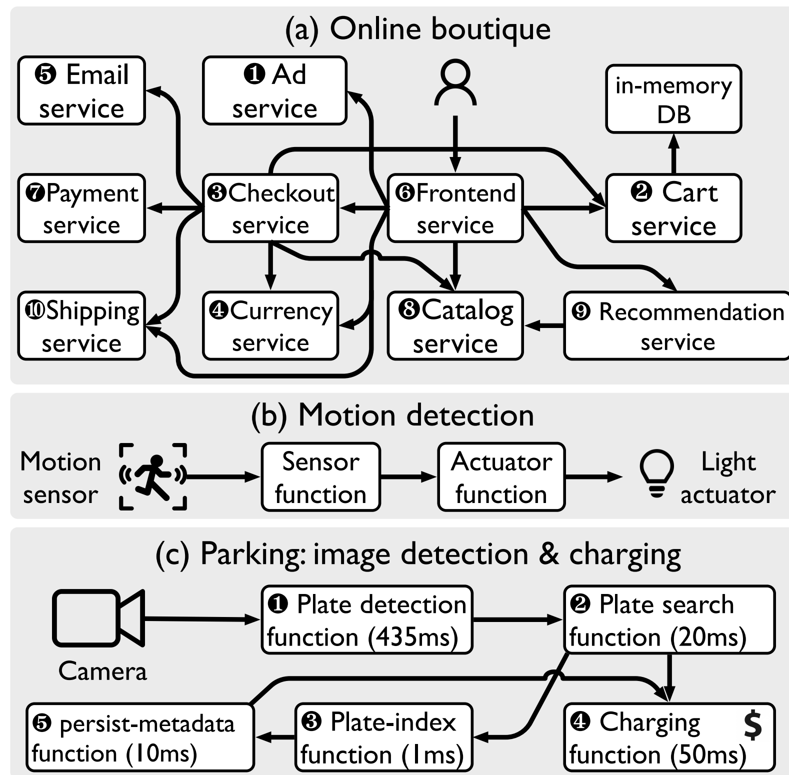


Figure 5.9: Serverless function chains setup. The Parking workload has two function chain invocation sequences: (Chain-1) ①→②→③→⑤→④; (Chain-2) ①→②→④

workload to send snapshot images (150×150 pixels, ∼3KB each) through HTTP/REST API call. Every 240-second interval, 164 snapshots are sent to the function chain. We use NGINX to coordinate the message exchanges within the chain. We use VGG-16 as the image detection algorithm, and the CPU service time of the image detection function is set to 435ms [100]. The CPU service times of other functions and the sequence of functions being called are shown in Fig. 5.9.

We use these serverless applications to quantify the performance gain brought by each of SPRIGHT's optimization. We evaluate it based on several metrics, including CPU usage, RPS, and response time. To understand in detail, we show the time series and CDF when appropriate.

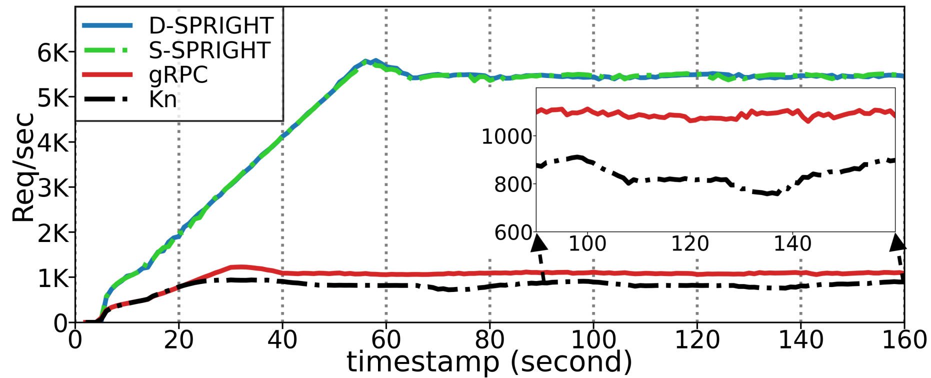


Figure 5.10: RPS for online boutique: {Knative, gRPC} at 5K & {D-SPRIGHT, S-SPRIGHT (overlap)} at 25K concurrency.

Testbed setup: The testbed is built on top of a base Knative platform, including 1) Knative serving/eventing components (v0.22.0) [61, 60]; 2) Kubernetes components (v1.19.0), including API server, placement engine, etcd, *etc* [42]. We use the docker engine (v20.10.21)

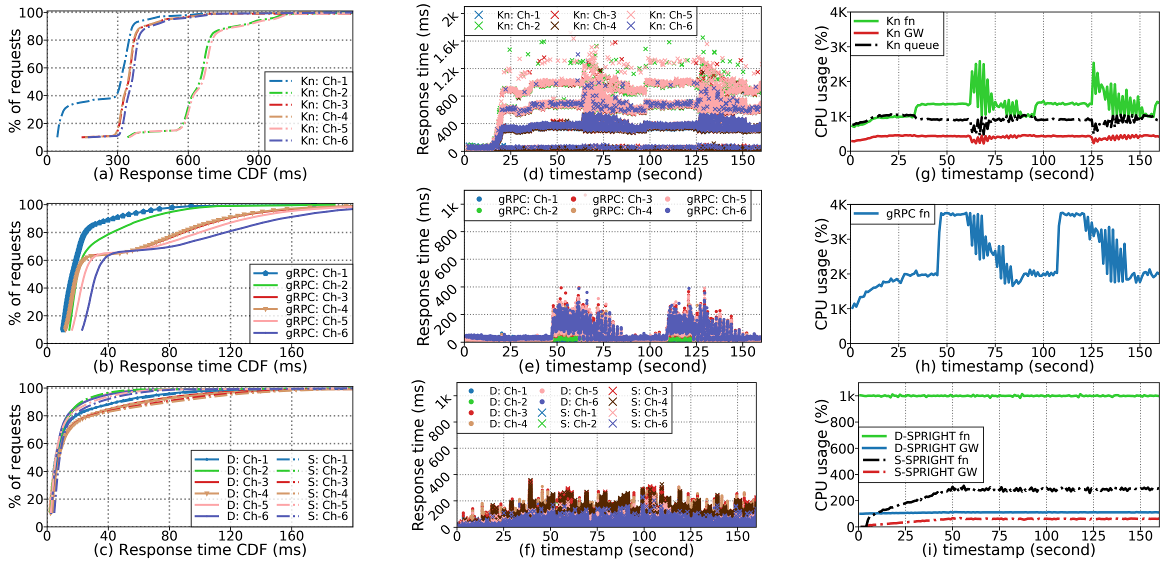


Figure 5.11: Online boutique. Top row: Knative, 5K concurrency. Mid. row: gRPC, 5K concurrency. Bottom row: {D-SPRIGHT (D), S-SPRIGHT (S)}, 25K concurrency. (Left col.) Response time CDF for 6 different function chains; (Mid. col.) Time series of response time, function chains; (Right col.) CPU usage time series, gateway (GW), function chains (fn), sidecar proxy (Knative)

as the container runtime. We consider Calico CNI (Native routing mode) [68] as the underlying networking solution except for the communication within the function chain of Knative. We run the experiments on the NSF Cloudlab with two c220g5 nodes [102]. Each node has a 40-core Intel CPU@2.2 GHz, 192GB memory, and a 10Gb NIC. We use Ubuntu 20.04 with kernel version 5.16. We configure the concurrency of both Knative and SPRIGHT function as 32. The concurrency level of a function pod determines the # of requests it can process in parallel.

5.4.2 Performance with Realistic Workloads

Comparing SPRIGHT, Knative, and gRPC mode We now compare D-SPRIGHT (using DPDK's RTE rings) and S-SPRIGHT (using SPROXY) against Knative and the gRPC mode for several different function chains of the online boutique application. We configure different concurrency levels (*i.e.,* # of concurrent users) of requests from the Locust load generator. We select two concurrency levels, 5K and 25K, to show here. To achieve the 5K concurrency, we set the spawn rate of 200/sec. concurrent requests. The spawn rate controls the # of concurrency steps increased every second. Above 5K, Knative's performance becomes highly variable with time, indicating overload (also results in very high tail response times). Both S-SPRIGHT and D-SPRIGHT have stable performance at a 25K concurrency level, after which they begin to show behavior indicating a slight overload. To achieve the 25K concurrency, we set the spawn rate of concurrency at 500/sec.

Even at 5K concurrency, Knative already begins to be overloaded. From 0s to 35s (Fig. 5.10), the concurrency level of the load generator is ramping up to 5K, and the requests/sec (RPS) increases to ∼900 req/sec. Knative begins to overload (see at 35s in

Fig. 5.10) due to the use of sidecars and the use of the Istio ingress gateway (hereafter referred to simply as 'gateway') to mediate the communication between functions. At this 5K concurrency, the gateway and sidecars consume ∼13 CPU cores (from 35s onwards), which is 50% of the entire Knative setup. It finally leads to CPU contention with the functions, whose CPU utilization soon reaches saturation at 62s (using up ∼13 CPU cores, Fig. 5.11 (g)). In addition, the use of the gateway and sidecars contributes to additional processing and queuing delays on the request's data path, leading to the reduction in RPS observed (see beyond 30s in Fig. 5.10). The closed-loop of workload generation and request processing results in the RPS, resource utilization, and response times experiencing overload cycles (occurs again between 100s - 140s).

Compared to Knative, gRPC has a more stable RPS and better overload behavior at 5K as gRPC has no sidecars and bypasses the gateway. By removing these heavyweight components, functions in the gRPC mode make full use of CPU resources. The shortened request data path further reduces latency and alleviates overload and queuing problems. As shown in Fig. 5.11 (a) and (b), the resulting tail latency of gPRC, *i.e.,* 95%ile, of 141ms, measured across all the functions of the online boutique service, which is 4.9× lower than Knative (whose 95%ile is 693ms). Fig. 5.11 (d) and (e) further demonstrate the benefits of removing sidecars and the gateway. For requests sent between 35s and 75s, the response time of Knative increases significantly while the gRPC shows a delayed overload (only 45s onwards) and its response time during the overload (45s to 75s) is much lower than Knative. However, as gRPC depends on the kernel protocol stack for networking and requires serialization/deserialization. These overheads are not negligible. The entire gRPC

setup consumes 91% of the total CPU cores available on the physical node in order to drain the queued requests (*e.g.,* 45s to 75s in Fig. 5.11 (h)). This pattern repeats again, *e.g.,* in the time period 108s - 140s. Overall, this is quite inefficient.

Table 5.3: Latency comparison at 5K and 25K concurrency

	Latency @ 5K (ms)			Latency @ 25K (ms)		
	95%	99%	Mean	95%	99%	Mean
Knative	**693**	**965**	**382**	-	-	-
gRPC	**141**	**199**	**45.6**	-	-	-
D-SPRIGHT	**11.1**	**45.1**	**5.8**	**80.8**	**144**	**17.7**
S-SPRIGHT	**13.4**	**49.2**	**7.2**	**96.1**	**159**	**20.0**

Note: latency is measured across all the functions of the online boutique.

Compared to Knative and gRPC, D-SPRIGHT and S-SPRIGHT both have stable RPS throughout the experiment, for concurrency levels ranging from 5K all the way to 25K. At 5K concurrency, The 95%ile latency of D-SPRIGHT and S-SPRIGHT are 11ms and 13ms (see Table 5.3), significantly less than Knative (690ms) and gRPC (140ms), while utilizing far less CPU. Although D-SPRIGHT constantly consumes CPU cycles when idle, even at maximum load, it consumes only 11 total CPU cores at a concurrency level of 5K, which is ∼2.5× less than Knative (similar to Fig. 5.8). This again validates the benefits of SPRIGHT's shared memory processing, saving CPU resources by avoiding the needless processing overheads with Knative discussed previously in §5.2. S-SPRIGHT *further* reduces CPU usage dramatically by using purely event-driven processing compared to D-SPRIGHT. With 5K concurrency, S-SPRIGHT consumes only ∼1 CPU core, including the gateway and all the functions, getting comparable performance (throughput, response time) to D-SPRIGHT. We further increase the concurrency level of the load generator to 25K for

D-SPRIGHT and S-SPRIGHT. This increases the utilization, but still maintains low tail response times. Both D-SPRIGHT and S-SPRIGHT maintain a stable RPS of ∼5500 req/sec (Fig. 5.10), which is 5× higher than the highest stable RPS achieved with Knative and gRPC. Moreover, S-SPRIGHT uses far less CPU resources than D-SPRIGHT, even as the load increases. At 25K concurrency, S-SPRIGHT consumes only ∼3.5 CPU cores, which is 3× less than D-SPRIGHT (Fig. 5.11(i)), showing the benefit of the eBPF-based event-driven processing.

With SPROXY generating context switches and interrupts for descriptor delivery (Table 5.2), there is some additional latency in S-SPRIGHT's shared memory processing, and is slightly worse than D-SPRIGHT in terms of tail latency (Fig. 5.11(c)). The 95%iles of S-SPRIGHT, measured across all the functions, is 1.2× higher than D-SPRIGHT (more details in Table 5.3). The additional delay for SPROXY's descriptor delivery, adds to the transient queueing and hence slightly longer tail latency. However, as we said in §5.3.3, the impact of this additional latency introduced by SPROXY is quite limited. Further, the processing time within the functions, which usually are non-trivial, will likely dwarf the extra latency introduced by SPROXY, in relative terms. Importantly, the throughput (RPS) of S-SPRIGHT is very close to D-SPRIGHT at high concurrency levels.

Bypassing the impact of cold start and zero scaling We set up an experiment with zero scaling enabled in Knative to study the impact of cold start. Without incoming requests, Knative scales functions down to zero to save resources and reduce costs. We set the 'grace period' for scaling down to zero as 30 seconds. In contrast, we keep functions

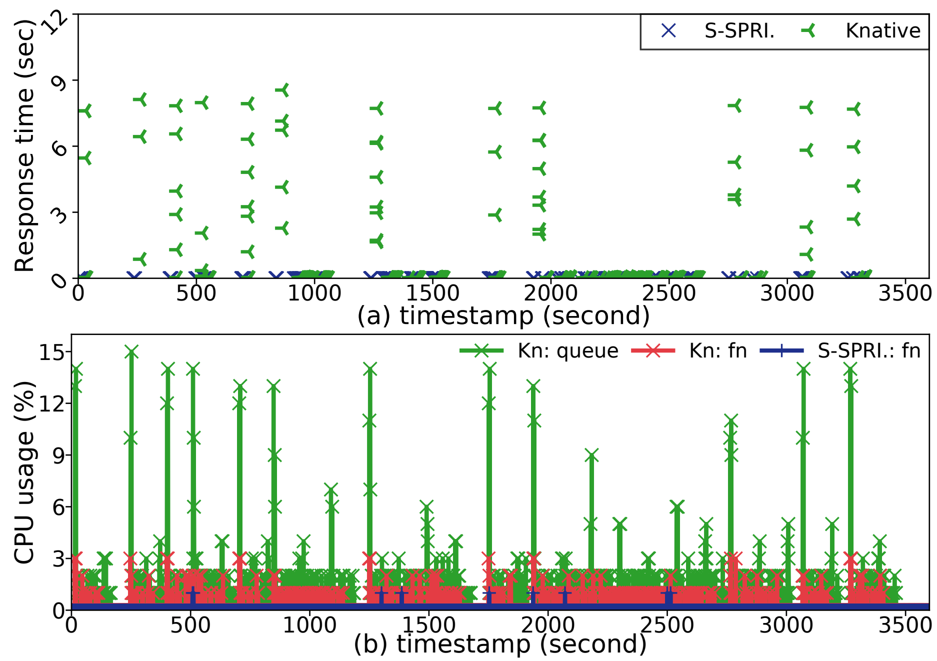


Figure 5.12: Time series of response time, and CPU utilization for motion detection workload - 1-hour long experiments.

in SPRIGHT 'warm' by having a minimum number of active function pods, knowing that our purely event-driven processing will not consume CPU resources when idle. We use the motion detection workload to study the impact of cold start because of the intermittent nature of such IoT traffic.

Fig. 5.12 (a) clearly shows the impact of cold start in Knative, with large response times that possibly render the motion detection application ineffective and severely violate SLOs. E.g., starting from 1950s, a number of motion events occur one after another (inter-arrival time of a few seconds) that are sent to the currently zero-scaled function chain. The first motion event that arrives at the gateway is queued and triggers the instantiation of the functions. Since a serverless function pod takes some time to start, subsequent requests have to be queued. The cascading effect during the cold start of the entire function chain further degrades the response time [173], resulting in a long tail latency going up to 9s. Once the function is active, Knative has a reasonably small response time when there are consecutive incoming events (*e.g.,* before the grace period terminates between 2000s and 2500s), which keeps the functions 'warm'.

In contrast, SPRIGHT shows consistently low response times over the entire workload duration since there is always an active pod to serve the request without leaving requests waiting in the queue (we can sidestep going down to zero-scale). More importantly, although SPRIGHT keeps one (or more) function warm, the event-driven nature of SPRIGHT leads to negligible CPU consumption when there is no traffic. In fact, with Knative, the higher resource usage of the sidecar proxy under load more than offsets any benefit of Knative's zero-scaling. E.g., in Fig. 5.12 (b), the spikes in the CPU usage for the

sidecar proxy (*e.g.,* at the 1500s mark), even when handling small traffic, is quite wasteful and is eminently avoidable with SPRIGHT's event-driven design.

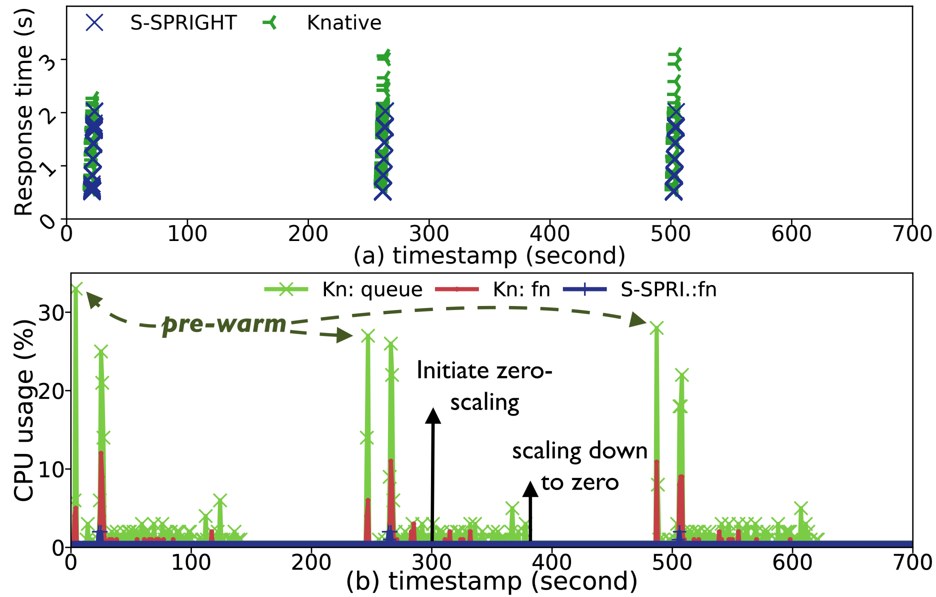


Figure 5.13: Parking image detection & charging: (a) Time series of response time of function chains; (b) Time series of aggregate CPU for function chains, sidecar proxy (Knative).

Since the 'Parking: image detection & charging' workload has a distinct periodic arrival pattern (*e.g.,* monitoring and billing every 4 minutes), we configure a 'pre-warm' phase for Knative functions 20 seconds before the next burst is scheduled to arrive. 'Pre-warming' helps avoid the penalty of the cold start delay of serverless functions while trading off a small amount of the resource savings of shutting down the pods in serverless computing with zero-scaling [196]. However, as observed in Fig. 5.13 (b), the CPU usage for each function instantiation at the pre-warming stage in fact exceeds the CPU usage consumed by request processing (*i.e.,* observe the CPU usage spike for the pre-warming and the function execution 20 seconds later). Thus, while zero-scaling reduces CPU usage if the idle

period is long, a CPU cost for frequent creation/destruction of functions must be considered. Knative also is quite inefficient for scaling functions down to zero. When there is no traffic for a grace period of 30s (*e.g.,* 270s to 300s in Fig. 5.13 (b)), Knative begins scaling down the functions to zero. But, functions remain in a 'terminating' state until 380s without being really terminated or releasing CPU resources. Thus, the scaling down process lasts as long as 80s, during which all the Knative sidecar proxies and functions are consuming CPU resources, which is unnecessary and wasteful.

For comparison, S-SPRIGHT consumes only a small amount of CPU throughout the entire period, in fact with slightly lower (about 16%) response time (both average and 95%, Fig. 5.13 (a)). Overall, S-SPRIGHT saves up to 41% CPU cycles in this 700s experiment without resorting to zero-scaling, almost doubling system capacity compared to Knative.

5.4.3 Startup latency comparison (SPRIGHT vs. Knative)

We now quantify the improvement in startup latency of the function pods in a chain with SPRIGHT compared to Knative. Depending on the control plane implementation and container runtime used by SPRIGHT and Knative, the time taken by the control plane activities and container runtime creation can be different for the two cases. To understand the overall startup latency of a function pod, we use the same control plane and container runtime for SPRIGHT and Knative. The common control plane components include Kubernetes's pod scheduler, controller manager, and API server. We use Docker engine as the container runtime for both.

We use the functions of online boutique [65] chain as an example to study the startup latency of function pods. We reuse the testbed setup in §5.4.1. To measure the latency of control plane activity, we take a timestamp as soon as the Kubernetes controller manager (on the master node) receives the pod creation request and take another timestamp when the *kubelet* (on the worker node) is signaled by the control plane for the pod initialization tasks, including the dataplane setup and container runtime creation, with the difference quantified as the latency for performing control plane activities.

To measure the dataplane setup latency (t_{dp}) of a Knative function pod, we take a timestamp when the *kubelet* sends the networking setup request to the Container Network Interface (CNI) and another timestamp when the *kubelet* receives an acknowledgment from the CNI indicating the successful setup of the function pod's dataplane. For the dataplane setup latency (t_{dp}) of a SPRIGHT function pod, we measure the latency of completing the initialization of the SPROXY (Fig 5.5) and the attachment to the shared memory pool.

For both Knative and SPRIGHT, we quantify the difference between the total pod initialization latency (t_{pod}) and the dataplane setup latency (t_{dp}) as the container runtime creation latency ($t_c = t_{pod} - t_{dp}$). Total pod initialization latency (t_{pod}) is measured as the duration starting from when the *kubelet* is informed by the control plane for the pod initialization until the *kubelet* detects the readiness of the function pod.

When profiling the startup latency of SPRIGHT function pods, we disable the CNI to avoid spending unnecessary latency on setting up kernel iptables and configuring veth devices, since SPRIGHT functions do not require these. We keep the CNI plugin enabled for the startup of SPRIGHT gateway pod as discussed in §5.3.5.

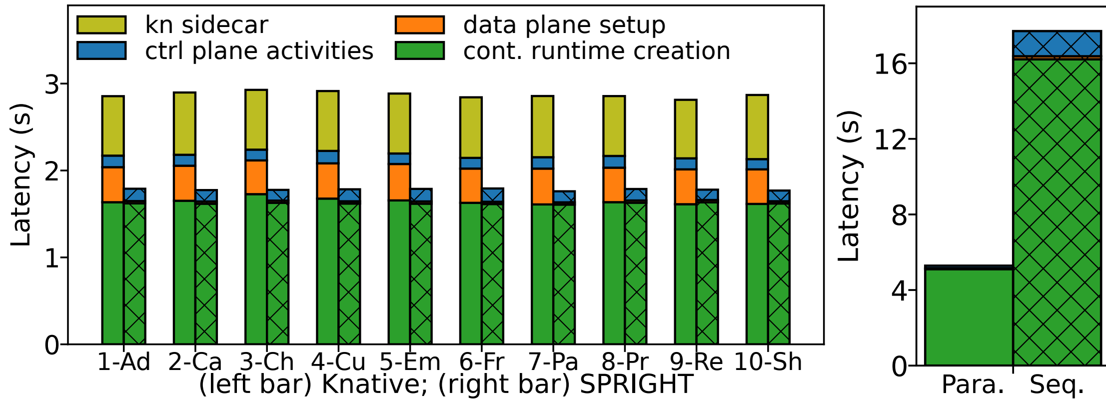


Figure 5.14: (Left) Latency comparison of a single function pod startup between SPRIGHT and Knative; and (Right) Latency of multiple function pods startup in SPRIGHT (startup in parallel VS. startup in sequence). For function's index, refer to Fig. 5.9.

Fig. 5.14 (Left) compares the startup latency of the individual functions of the online boutique for Knative vs. SPRIGHT starting up a single function pod at a time, sequentially. The latency spent on control plane activity is the same for both SPRIGHT and Knative. This is not surprising, as both use the same control plane components and policies. In addition, there is no significant latency difference for container runtime creation between SPRIGHT and Knative, as they both use the same container runtime (Docker). The primary difference is in the dataplane setup latency. Knative spends ∼0.4s for setting up the dataplane for a single function pod, while SPRIGHT takes only ∼0.016s for both SPROXY initialization and shared memory pool attachment. In addition, Knative needs to create an individual sidecar container for the function pod, which takes another ∼0.7s. This, again, shows the benefit of SPRIGHT's use of shared memory and event-driven proxies, which eliminates the heavyweight sidecar container creation and slow kernel-based dataplane setup.

Since each SPRIGHT function chain has a dedicated SPRIGHT gateway for consolidated protocol processing, there is an initialization overhead for the SPRIGHT gateway, the first time a SPRIGHT function chain is started. Compared to a SPRIGHT function pod, the SPRIGHT gateway pod takes a longer time to set up the dataplane. As the SPRIGHT gateway pod depends on kernel network stack to perform protocol processing, it needs to set up the veth-pair and kernel iptables via CNI, similar to a Knative function pod, which takes around ∼0.4s. Further, the SPRIGHT gateway pod takes an additional ∼0.3 milliseconds to attach the EPROXY to the veth-pair, which is negligible compared to the overall startup latency. But, the startup overhead of the SPRIGHT gateway can be overlapped by starting it in parallel with the SPRIGHT functions in the chain. More importantly, the startup of the SPRIGHT gateway is a one-time task that occurs only during the first startup of the function chain. The subsequent startup activities of functions in the chain do not incur the startup latency for the SPRIGHT gateway.

Fig. 5.14 (Right) compares the startup latency of a complete function chain involving the startup of multiple function pods in parallel. We compare this against starting each function of the chain sequentially. We use the online boutique with a total 10 function pods as an example. Starting up pods in parallel takes 3.3× *less* time than starting up pods sequentially. Thus, when starting up a function chain in SPRIGHT, it is desirable to have function pods started in parallel to amortize the startup penalty. Looking into the latency breakdown of the function chain startup in SPRIGHT, both dataplane setup in SPRIGHT and control plane activity get the benefit of parallelism. Having concurrent dataplane setup operations and control plane activity does not contribute additional la-

tency, and they have negligible impact on the overall startup latency. The dominant factor in the latency for starting up multiple pods is in the container runtime creation. When multiple container runtime creations are processed in parallel, the reduction in latency is limited because of contention for access to the network namespace in the Linux kernel. This forces us to sequentially create and modify the network namespaces [171, 209, 132], which contributes to the still relatively large time taken for the container runtime creation. We believe there is further room for improving the total container runtime creation time further. Since SPRIGHT's trust model assumes functions in the same chain share the same security domain (§5.3.7), a shared network namespace for the function pods within the same chain can seamlessly work with SPRIGHT's security domain design. This can eliminate the network namespace contention and reduce the container runtime creation latency for the function pods within the same chain.

5.5 Conclusion

SPRIGHT demonstrated the effectiveness of event-driven capability for reducing resource usage in serverless cloud environments. With extensive use of eBPF-based event-driven capability in conjunction with high-performance shared memory processing, SPRIGHT achieves up to 5× throughput improvement, 53× latency reduction, and 27× CPU usage savings compared to Knative when serving a complex web workload. Compared to an environment using DPDK for providing shared memory and zero-copy delivery, SPRIGHT achieves competitive throughput and latency while consuming 11× fewer CPU resources. Additionally, for intermittent request arrivals typical of IoT applica-

tions, SPRIGHT still improves the average latency by 16% while reducing CPU cycles by 41%, when compared to Knative using 'pre-warmed' functions. This makes it feasible for SPRIGHT to support several 'warm' functions with minimum overhead (since CPU usage is load-proportional), sidestepping the 'cold-start' latency problem. Across several typical serverless workloads, SPRIGHT shows higher dataplane performance while avoiding the inefficiencies of current open-source serverless environments, thus getting us closer to meeting the promise of serverless computing. In addition, SPRIGHT saves 32% startup latency for a single function pod compared to Knative, which is an ideal capability for serverless computing. SPRIGHT is publicly available at https://github.com/ucr-serverless/spright.git

Chapter 6

LIFL: A Lightweight, Event-driven Serverless Platform for Federated Learning

6.1 Introduction

Federated Learning (FL [162]) enables collaborative model training across a network of decentralized devices/machines while keeping individual user data secure and private. In FL, instead of sending raw data to a central server, models are trained on individual devices/machines using local data, and only the model updates are shared and aggregated to create a global model.

To support FL at scale, *hierarchical aggregation* is often adopted to increase the service capacity for model aggregation [87, 133]. This can accommodate a large number

of clients and handle a substantial volume of model updates, avoiding potential slow-down of the aggregation process. In the process, each level performs intermediate aggregation, combining the updates from lower-level aggregators or clients.

Existing FL frameworks (*e.g.,* Google's FL stack [87], Meta's PAPAYA [126]) adopt a static, always-on[1] deployment to support model aggregation. However, in a dynamic FL environment, it's difficult to have a one-size-fits-all service capacity for model aggregation. System heterogeneity (*i.e.,* different hardware capabilities) and a dynamically varying number of participating clients in each round require frequent adjustments of the capacity so that the aggregation service effectively uses resources on demand and avoids significant resource wastage.

Serverless computing promises to provide an event-driven, resource-efficient cloud computing environment, enabling services to use resources on demand [196]. Running FL model aggregation service as serverless functions can right-size the provisioned resources and reduce resource waste compared to an always-on aggregation server implementation. In addition, stateless processing by serverless functions makes it easy to support continual updates to the aggregation hierarchy. By increasing the capacity of aggregation through a hierarchy of serverless aggregators, model aggregation in FL can be executed in parallel, responding to increasing loads from trainer model updates.

However, the excessive overhead in current serverless frameworks, caused by the loose coupling of data plane components [181], is a barrier to achieving efficient and timely aggregation, compared to a monolithic serverful design. Further, the use of individual, constantly-running components (e.g., container-based sidecars) in current serverless frame-

[1]meaning that aggregators are up all the time within a round.

works is inefficient and sacrifices much of the benefit of serverless computing. This prompts us to create a more streamlined, responsive serverless framework that is tailored to achieve just-in-time FL aggregation on demand.

We introduce LIFL [183], a lightweight serverless platform for FL that uses hierarchical aggregation to achieve parallelism in aggregation and exploits intra-node shared memory processing to reduce data plane overheads. LIFL also utilizes a locality-aware placement policy to maximize the benefits of the intra-node shared memory data plane. Unlike typical serverless platforms that use a heavyweight sidecar implemented as a separate container, LIFL seeks to eliminate this wasteful overhead by taking advantage of eBPF-based event-driven processing. This ensures that resource usage is truly load-proportional. Instead of depending on inaccurate, threshold-based autoscaling, LIFL uses hierarchy-aware autoscaling to precisely adjust the capacity of model aggregation to match the incoming load. We also use a policy of reusing runtimes to sidestep the impact of startup delay on the model convergence time, while also improving resource efficiency of aggregation. LIFL favors eager aggregation to enable timely aggregation, reducing the queuing time for model updates. By harnessing the capabilities of LIFL, FL systems can achieve efficient resource utilization and reduced aggregation time. LIFL is available at [57].

We highlight the contributions of LIFL below:

(1) LIFL's enhanced data plane achieves 3× (compared to serverful) and 5.8× (compared to serverless) latency reduction on transferring a relatively heavyweight ResNet-152 model update within the aggregation hierarchy (intra-node).

(2) LIFL's locality-aware placement can maximize shared memory processing, achieving up to 2.1× additional latency reduction on aggregating a batch of updates in a round (details in §6.6). After applying hierarchy-planning, aggregator reuse, and eager aggregation, LIFL can further obtain 1.5× latency reduction. The enhanced orchestration also helps improve efficiency, saving up to 2× CPU consumption compared to simply using the enhanced data plane.

(3) Our evaluation with a real FL workload using ResNet-18 and 120 simultaneous active clients (the total number of clients used is 2,800) shows that the combination of LIFL's enhanced data and control planes achieve **5**× and **1.8**× less CPU cost and reduces **2.7**× and **1.6**× on time-to-accuracy (70% accuracy level), compared to existing serverless and even serverful FL systems. We also train a relatively heavyweight ResNet-152 model. LIFL spends 1.68× less time to reach 70% accuracy than existing serverless FL systems, while using 4.23× fewer CPU cycles.

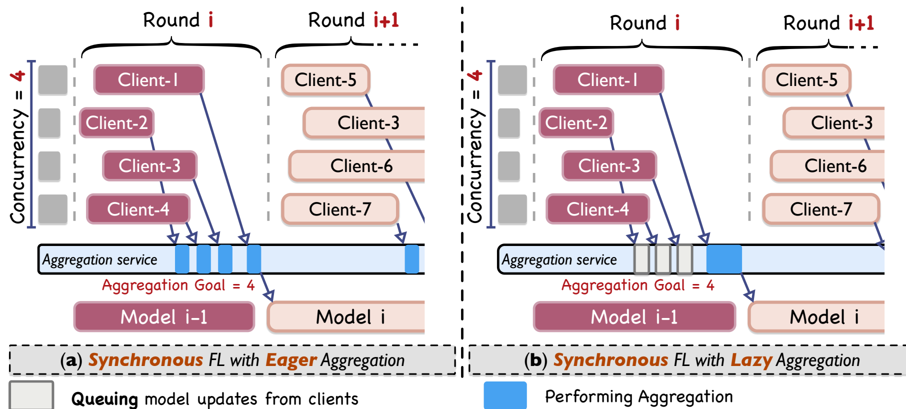


Figure 6.1: Synchronous FL with different aggregation timing ("Eager" and "Lazy") [87, 134].

6.2 Background and Challenges

6.2.1 Basics of Federated Learning

FL aggregation: Aggregation in FL is a process of building a global model from individually trained model updates. The *aggregation goal*, n specifies the expected number of model updates to be received before the global model is updated to a new version. Thus, it dictates the number of selected clients for training. The aggregation process is abstracted as:

$$w_i = f(\{(w_i^k, \mathcal{A}_i^k) \mid 1 \leq k \leq n\}). \tag{6.1}$$

Here $f(\cdot)$ is an aggregation function, w_i^k is k-th local model update for global model version i, and $\mathcal{A}_i^k$ is auxiliary information for aggregation. For the *FedAvg* algorithm [162], $f(\cdot) = \sum_{k=1}^{n} w_i^k c_i^k / T_i$. $T_i = \sum_{k=1}^{n} c_i^k$ and $\mathcal{A}_i^k$ is c_i^k (the number of data samples).

Eager aggregation and Lazy aggregation: Based on the timing to trigger the aggregation, we can classify the model aggregation to be "*eager*" or "*lazy*" [134]: *Eager* aggregation allows aggregation to happen whenever an update is received, leading to more flexible and dynamic timing of the aggregation process. *Lazy* aggregation operates on a delayed schedule, where model updates that arrive early are queued without being aggregated immediately. Fig. 6.1 shows the two different aggregation methods for synchronous FL. For instance, the *eager* method is feasible for FedAvg with cumulative averaging.

6.2.2 Anatomy of Systems for Federated Learning

Designing a system to support FL at a ***large scale*** is essential, as a larger number of participants means a more diverse and representative dataset. It improves the model's ability to capture complex patterns and unseen relationships in the data. These benefits help the model generalize in real-world deployments, *e.g.,* Google's FL stack has been used to serve $\sim 10M$ devices daily and $\sim 10K$ devices participate in FL training simultaneously [87].

Fig. 6.2 depicts key architectural components that are needed to ensure the success of FL at scale.[2] These components work together to enable the collaborative and decentralized training process in FL. In addition to the **aggregator** and the **client**, the **coordinator** oversees the flow of FL operations. It acts as an orchestrator that facilitates seamless interactions among aggregators, selectors, and clients by applying the client selection scheme and instructing the selector to map the selected clients to backend aggregators [87]. The **selector** plays two roles. First, it ensures that a diverse set of clients participate in the FL process to capture a representative sample of the distributed data. Second, it acts as a gateway that mediates communication (*i.e.,* queuing, load balancing) between (leaf) aggregators and clients [87, 126].

Need for hierarchical aggregation: The growing number of participating clients in FL requires the system to be scalable to accommodate the computational requirements of aggregating model updates from a large number of distributed clients. This primarily motivates the use of *hierarchical aggregation* potentially involving multiple levels of aggregation in the FL process [87, 133], as depicted in Fig. 6.2 (a). Essentially, hierarchical aggregation is

[2] We adopt the terminology of FL system components from [87] and [126].

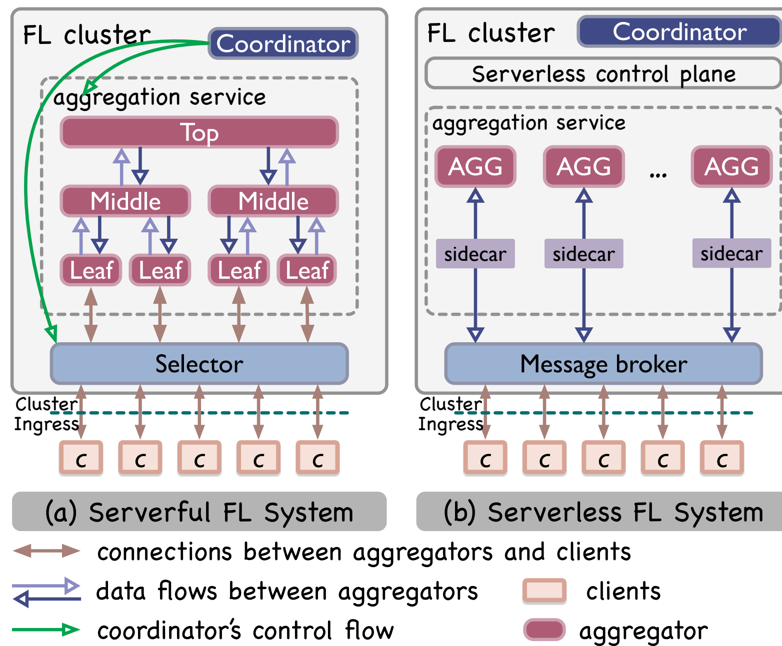


Figure 6.2: Generic architectures for FL systems: (a) Serverful FL systems [87, 126]; (b) Serverless FL systems [135, 133, 94]. Note that for simplicity, we skip the hierarchy in the diagram (b).

structured as a single-rooted tree. Each level in the tree includes multiple parallel aggregation tasks that are executed by one of potentially multiple aggregators. The communication during the hierarchical aggregation task takes place across multiple levels: The model updates from smaller subgroups of clients are aggregated by the lower-level aggregators (*i.e.,* leaf) and passed onto higher-level aggregators (*i.e.,* top), until a global model is obtained. This parallel aggregation at the lower levels can provide speedup and reduce queueing of model updates.

6.2.3 Motivation and Challenges for Serverless FL

State-of-the-art FL systems [87, 126] rely on a "serverful" design that relies on a fixed pool of dedicated resources (*e.g.,* CPU and memory), using a pool of provisioned VMs. Resizing the pool often takes a long time (e.g., 6 to 45 minutes on AWS [194]). Serverless computing, on the other hand, brings fine-grained resource elasticity by provisioning functions (typically as containers) dynamically based on demand, ensuring that the right amount of resources is allocated only when needed.

In FL, serverless computing can be used to provide efficient model aggregation, adapting to varying numbers of clients. It eliminates the need to maintain dedicated resource pools for the aggregation service, thereby improving overall efficiency compared to the current "serverful" deployments.

Prior Work on Serverless FL. A number of FL system designs have been proposed using serverless computing [135, 133, 116]. A common abstract architecture of a serverless FL system and its key components is shown in Fig. 6.2 (b). But, prior approaches still face the following challenges:

Indirect networking: Unlike a "serverful" design (Fig. 6.2 (a)), a serverless FL system executes aggregators as serverless functions. Serverless function chaining can support hierarchical aggregation as well as communication between aggregators. However, because serverless functions are ephemeral and stateless (and thus unable to retain stateful information like routes), these chains typically only support *indirect* networking between functions. This raises the need for a stateful, persistent networking component (Fig. 6.2 (b)), such as a message broker or external storage services,[3] to maintain routes and exchange messages [181]. However, having such a networking component in the internal datapath between serverless functions adds unnecessary overhead (20% added delay as in Fig. 6.9(a)).

Inefficient message queuing: In addition to supporting function chaining, the message broker (Fig. 6.2 (b)) also acts as a message queue to buffer incoming model updates from clients while aggregators are being spawned by the serverless control plane [135, 133]. However, the message broker and dedicated queues add overhead and delay to the aggregation service.

Heavyweight sidecar: Scheduling serverless functions typically requires metrics collection, often using a sidecar. This container-based sidecar introduces additional network processing in the datapath, requiring the interception and forwarding of model updates. This leads to complex data pipelines (involving extra communication hops between aggregators) and increased communication overheads due to the reliance on kernel-based networking [181].

[3]For consistency, we use the generic term "message broker" to denote such a networking component throughout this paper.

Application-agnostic, simple, autoscaling: Current serverless autoscaler designs typically rely on a simplistic threshold based on user input (*e.g.,* request per second, concurrency) for scaling decisions [37, 36], often being unaware of application needs. This design, agnostic of the hierarchical structure of FL aggregation, is limited in its ability to optimize the system to maximize parallelism, *i.e.,* the number of levels and the number of aggregators at each level. Looking at Fig. 6.2 (a), as we go up the levels in the hierarchy, fewer aggregators are needed. This can be leveraged to potentially reuse the lower-level aggregators as we proceed up the hierarchy. Further, since hierarchical aggregation uses function chaining, current "reactive" autoscaling designs lead to a cascading effect [173] of the cold-start delays when scaling a function chain.

Locality-agnostic placement: *Intra-node* communication can be faster than *inter-node* communication by avoiding a lot of networking overheads [178] and using state-of-the-art serverless data plane designs with *shared memory* [197, 181, 223]. However, leveraging the benefits of shared memory effectively can be challenging when dealing with a large hierarchy of aggregators that cannot be accommodated within a single node. This requires careful function placement by taking into account the impact of communication between aggregators. Inter-node communication typically still uses kernel-based networking.

6.3 LIFL Overview

We aim to address the aforementioned limitations (§6.2.3) and develop LIFL—a high-performance, lightweight, and elastic serverless platform for FL, utilizing hierarchical aggregation. We focus on the following innovations of LIFL:

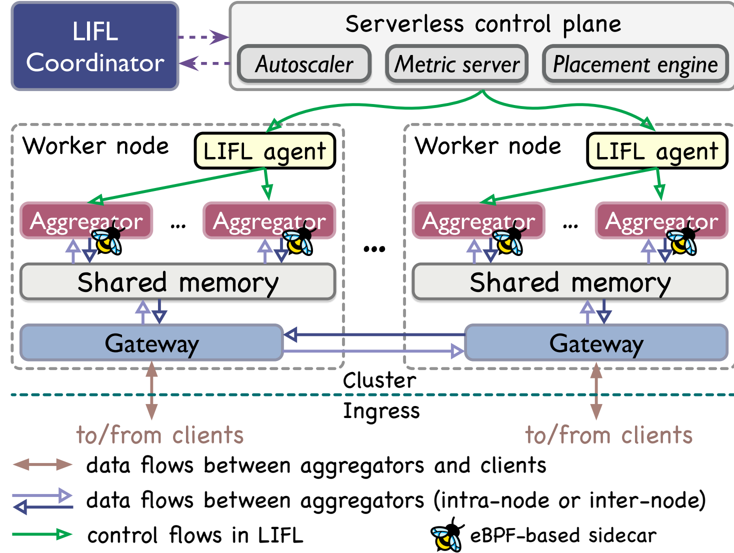


Figure 6.3: The overall architecture of LIFL.

(1) High-performance intra-node dataplane: LIFL incorporates shared memory processing to provide a *zero-copy* communication channel between FL aggregators placed on the same node (§6.4.1). This avoids heavyweight kernel networking overheads, especially data copies [90], as model updates are often large, *e.g.,* a model update from ResNet-152 [124] is ∼230 MBytes. Shared memory can also eliminate other overheads such as protocol processing, serialization/de-serialization, kernel/userspace boundary crossing, and interrupts.
(2) In-place message queuing: We extensively leverage shared memory in LIFL to offer "in-place" message queuing (§6.4.2). Messages (*i.e.,* model updates) from selected clients are directly buffered in shared memory and can be instantly accessed by the aggregators when they are ready. This eliminates dedicated message queues and their associated queuing delays.
(3) Lightweight eBPF-based sidecar: We incorporate the extended Berkeley Packet Filter (eBPF [39]) into LIFL to build a lightweight sidecar (§6.4.3) to provide important

functionality, *e.g.,* metrics collection. Unlike a container-based sidecar, LIFL's sidecar runs as eBPF code attached at in-kernel hooks, avoiding the need for dedicated resources. We further utilize the eBPF-based sidecar to support *direct* networking between aggregators (§6.4.4), completely replacing the message broker.

(4) A cost-effective orchestration heuristic: LIFL orchestrates the model aggregation to fully exploit the improved serverless dataplane by employing several strategies: (1) locality-aware placement that partitions levels with large traffic into node-affinity groups to make the best use of shared memory processing (§6.5.1); (2) hierarchy-aware scaling that dynamically adjusts the configuration of hierarchical aggregation (§6.5.2), and (3) opportunistic reuse of the aggregator runtime from a lower level (§6.5.3).

Architectural overview of LIFL: Fig. 6.3 shows the overall architecture of LIFL. LIFL maintains a shared memory object store on each worker node to enable zero-copy communication between aggregators. To support in-place message queuing, LIFL introduces a gateway on each worker node that receives model updates from remote clients. The gateway performs a consolidated, one-time payload processing to queue the received model updates to shared memory. Each aggregator in LIFL has attached to it an eBPF-based sidecar for lightweight metrics collection. Aggregators in LIFL are stateless, so new ones start without state synchronization upon an aggregator failure. LIFL detects client failures with keep-alive heartbeats and enhances resilience by over-provisioning the number of clients. In the control plane, a LIFL agent is deployed on each worker node to manage the lifecycle (*e.g.,* creation, termination) of aggregators, following instructions from the LIFL control plane. The LIFL coordinator, a cluster-wide control plane component, is used for interactions between the FL job designer (ML engineer) and the serverless control plane

(*e.g.,* autoscaler, placement engine).[4] It works with the serverless control plane to execute LIFL's orchestration flow (§6.5).

6.4 Optimizing the Serverless Data-Plane in LIFL

6.4.1 Shared Memory for Hierarchical Aggregation

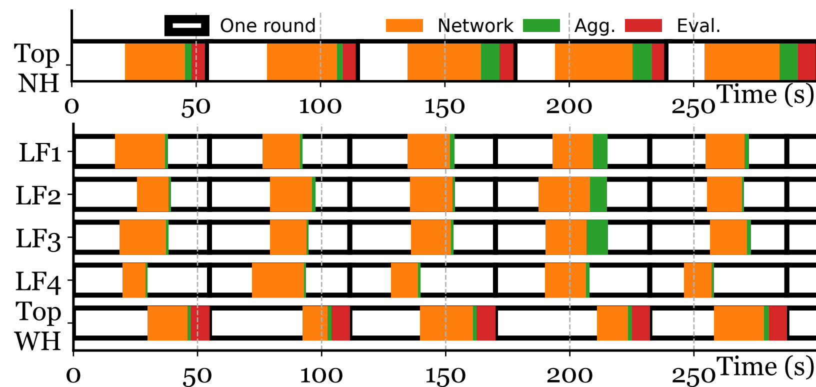


Figure 6.4: Impact of data plane performance on hierarchical aggregation. (upper fig.:) No hierarchy(NH); (lower fig.:) With hierarchy(WH). Top: top aggregator; LF: leaf aggregator. "Network" denotes the data transfer tasks of model updates; "Agg." denotes the aggregation tasks; "Eval." denotes the evaluation tasks.

Assessing data plane with hierarchical aggregation: We now assess the importance of a high-performance data plane to truly deliver on the promise of hierarchical aggregation. We consider a baseline (denoted `NH`) with a single aggregator without hierarchy. We evaluate the hierarchical aggregation service that has one top aggregator and four leaf aggregators

[4]Note: Even though the serverless control plane has serverful, always-on components (*e.g.,* autoscaler, placement engine), are shared and their overheads are amortized across multiple workloads, especially at scale.

(denoted `WH`). All aggregators are placed on the same node. We consider eight trainers to train a ResNet-152 model using FEMNIST dataset. Note that we always deploy trainers on separate nodes, to both be realistic (trainers are remote) and to avoid contention for resources on the node.

Fig. 6.4 shows the execution times for the representative FL stages under different settings. Note that we only show the receiving part of the networking task ("Network" in Fig. 6.4) to simplify the figure. Compared to the baseline (`NH`), `WH` does not exhibit a significant improvement overall, though it uses hierarchical aggregation. The average completion time per round with `WH` is 57 seconds, while for `NH` is 59.8 seconds. This is mainly because of the contention for network processing between leaf aggregators when they send/receive intermediate model updates to/from the top aggregator. This highlights the critical need for a high-performance and streamlined data plane for hierarchical aggregation. LIFL incorporates shared memory processing when the serverless aggregator functions are co-located on the same node. This enables fast and efficient communication, mitigating the impact of networking on hierarchical aggregation (demonstrated in Fig. 6.9). Working jointly with our locality-aware placement scheme (§6.5.1), LIFL can minimize the need for inter-node model update transfers. Consequently, LIFL maximizes the advantages of our efficient intra-node shared memory data plane that substantially reduces communication overheads.

Shared memory object store: The LIFL agent is responsible for the allocation, recycling, and destruction of the shared memory buffer in the object store. In addition, LIFL only allows immutable (read-only) objects to guarantee the safe sharing of model updates,

eliminating the need for locks. The agent periodically checkpoints the model parameters to an external persistent storage service to ensure data persistence and potential recovery in case of failures. The checkpointing occurs after the aggregator completes the aggregation of specified model updates, where the aggregator submits a request to the LIFL agent to perform model checkpoints asynchronously in the background. This prevents checkpoint delays from being added to the aggregation completion time.

6.4.2 In-place Message Queuing

Representative message queuing solutions: Fig. 6.5 enumerates message queuing solutions for various serverful and serverless alternatives. In the *monolithic* serverful setup (used in [126]), the model update is directly buffered into an in-memory queue residing in the aggregator, deployed as a persistent and stateful application. Another serverful setup, used in [87], deploys aggregators as ephemeral, stateless *microservices*, requiring an additional persistent, stateful message broker to buffer model updates from clients before being consumed by the stateless aggregator. Switching to the *basic* serverless setup (used in [133]), model updates are also buffered at a message broker, as the aggregator is now deployed as an ephemeral, stateless serverless function. Before being consumed by the aggregator, the model update has to pass through the sidecar. Finally, in LIFL, the gateway buffers the model update directly into the shared memory, which can then be seamlessly accessed by the aggregator. The distinct data pipelines between the client, message queue, and aggregator impose varying degrees of overheads. Our evaluation (details in §6.6.1) shows that LIFL's in-place message queuing achieves the best efficiency and performance (equivalent to a monolithic, serverful design) among all alternatives in Fig. 6.5.

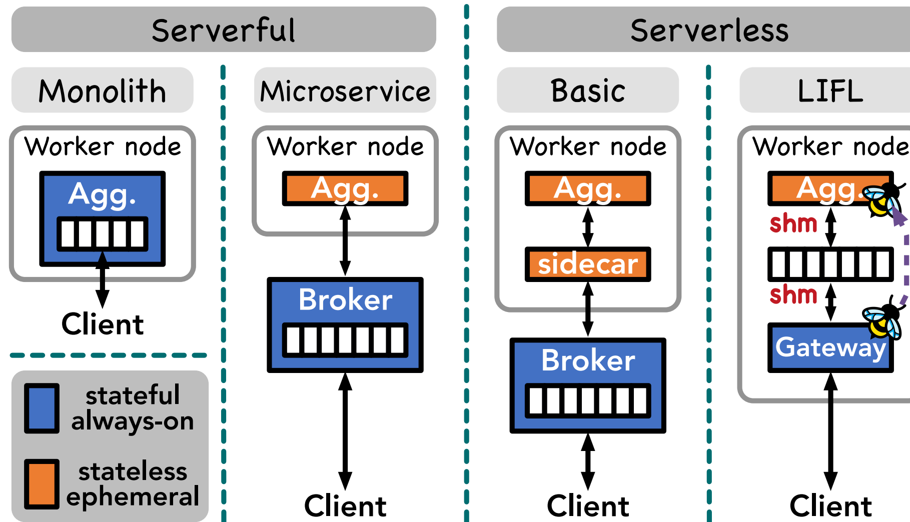


Figure 6.5: Message queuing solutions.

Message queuing pipeline in LIFL: The gateway at each worker node is addressable/accessible by FL clients. It receives model updates from clients or from the gateway on another worker node, and performs necessary network processing (*e.g.,* protocol processing, serialization, deserialization, data type conversion, *etc*) before writing the model updates into shared memory. This avoids duplicate processing when local aggregators access model updates in shared memory.

On the receive (RX) path, protocol processing by the kernel TCP/IP stack is first performed. The gateway running in userspace receives the raw L7 payload from the kernel and then extracts the model updates (encoded as `tensor` data type), depending on the adopted L7 protocol (*e.g.,* gRPC, MQTT). We convert the model update from `tensor` data type to `NumpyArray` before writing it to shared memory, as Python's `multiprocessing` module does not support manipulation of the `tensor` data type. On the transmit (TX) path, the reverse payload processing is done.

We apply vertical scaling of the gateway by dynamically adjusting the number of assigned CPU cores based on the load level. This avoids the gateway becoming the dataplane bottleneck and impacting the aggregation speed.

6.4.3 eBPF-based Sidecar

LIFL's eBPF-based sidecar is built with a set of eBPF programs attached to each aggregator's socket interface, using its in-kernel `SKMSG` hook [188]. The execution of the eBPF-based sidecar is triggered by the invocation of the *send()* system call, which is captured by the `SKMSG` hook as an eBPF event. This ensures that the eBPF-based sidecar is strictly event-driven and consumes *no* CPU resources when idle. We use the eBPF-based sidecar to collect necessary metrics (*e.g.,* execution time of the aggregation task) to facilitate the orchestration in LIFL (§6.5).

Metrics collection: Upon invocation, the eBPF-based sidecar collects and stores metrics to an eBPF map (*metrics map*) on the local worker node. The eBPF map is an in-kernel, configurable key-value table that can be accessed by the eBPF program during execution [38]. The LIFL agent, on the other hand, periodically retrieves the latest metrics from the *metrics map* and feeds the metrics back to the metrics server (Fig. 5.3) in the serverless control plane.

6.4.4 Direct Routing with Hierarchical Aggregation

Direct networking between functions is not allowed in existing serverless environments because serverless functions are considered to be stateless and ephemeral. This implies that there are no long-lived, direct connections between a pair of function instances.

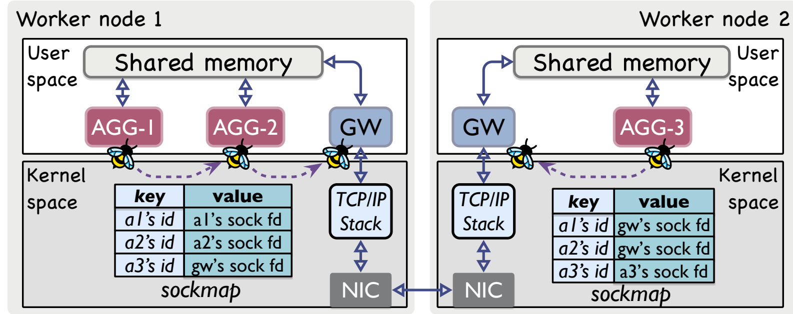


Figure 6.6: Intra-/inter-node direct routing within hierarchical aggregation.

As a result, they use an intermediate networking component (*e.g.,* message broker) to act as a stateful, persistent component to manage state, *i.e.,* routes between functions. However, the main drawback is that it adds unnecessary overhead by involving the additional networking component(s) in the datapath, making indirect networking between functions heavyweight.

LIFL improves serverless networking within hierarchical aggregation by allowing direct routing between aggregators, both within a node and between nodes. The key is to offload the stateful processing to eBPF, using the *sockmap* [188] to support flexible intra-node routing exploiting shared memory, and inter-node routing with the help of the per-node gateway, as depicted in Fig. 6.6. The *sockmap* is a special eBPF map (`BPF_MAP_TYPE_SOCKMAP` [188]) that maintains references to the registered socket interfaces. We take the approach from [181] to implement intra-node direct routing in LIFL, as described below.

Intra-node routing: LIFL makes full use of its shared memory support to facilitate *zero-copy* exchange of model updates between aggregators. The shared memory object in LIFL is addressed by the object key, which is a 16 byte string randomly generated by the shared memory manager when it initializes shared memory objects. We also assign each aggregator

a unique ID. The zero-copy data exchange between aggregators depends on delivering the object key, as the data is kept in place in shared memory.

LIFL utilizes eBPF's `SKMSG` (integrated in the eBPF-based sidecar), combined with eBPF's sockmap [188], to pass the object key between aggregators on the same node. Upon receiving the object key, the `SKMSG` program uses the ID of the source aggregator as the key to look up the *sockmap* to find the socket interface of the destination aggregator so that the object key may be delivered to it for access of the shared memory object.

Inter-node routing: When the source aggregator communicates with a destination aggregator on a different node, it sends the object key to the local gateway first. The local gateway uses the object key to retrieve the model update from shared memory and performs the necessary payload transformation. It then uses the source aggregator ID to look up the inter-node routing table to obtain the destination aggregator ID and the IP address of the remote node hosting the destination aggregator. The model update is sent through the remote node's gateway to the destination aggregator. The remote gateway stores the received model update in shared memory and uses `SKMSG` to notify the destination aggregator, along with the local object key.

Online hierarchy update: LIFL re-configures intra-/inter-node routes each time the hierarchy is updated. The routing manager in the LIFL agent takes the DAG input (generated by the TAG, §6.4.5) from the control plane that describes the connectivity between aggregators, and correspondingly updates routes into the inter-node routing table in the gateway and in-kernel *sockmap*, using the userspace eBPF helper, `bpf_map_update_elem()` [48]. The TAG describes the cross-level data dependency between aggregators.

6.4.5 Abstraction for fine-grained control

To facilitate fine-grained control of LIFL's orchestration, we treat an aggregator process within a sandboxed runtime (*e.g.,* container) as the atomic unit for management. The control plane needs a generic means to describe connectivity between components and placement affinity. We make use of Topology Abstraction Graph (*TAG*) in Flame [98] to describe the aggregator-to-aggregator connectivity and aggregator-client connectivity. Each node in such a graph is associated with a "*role*" metadata, denoted as either aggregator or client. A "*channel*" metadata denotes the underlying communication mechanism (*e.g.,* intra-node shared memory, inter-node kernel networking) used for connectivity.

We configure the placement-affinity to facilitate locality-aware placement through the *groupBy* attribute in the channel abstraction, which accepts a string as a label to specify a group. Therefore, keeping the same label in the attribute allows us to cluster roles into a group. The LIFL coordinator enables necessary orchestration decisions, *e.g.,* runtime reuse and locality-aware placement, through manipulation of these abstractions (role and channel).

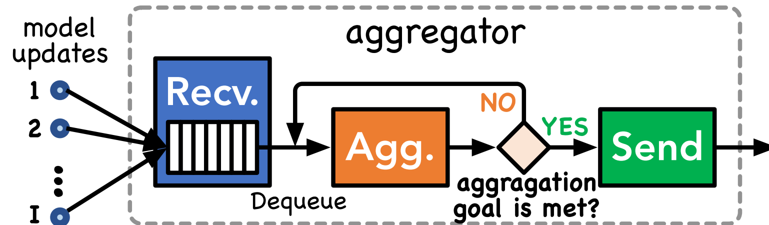


Figure 6.7: Step-based processing model.

6.4.6 Step-based Processing Model

The basic processing model of an LIFL aggregator can be abstracted as a multiple-producer, single-consumer pattern, as shown in Fig. 6.7. Multiple upstream producers (clients or aggregators) are mapped to a single consumer (aggregator only). The single consumer gathers model updates from assigned producers and computes the aggregated model update.

Looking deeper into the aggregator, LIFL adopts a step-based processing model. At the core of this design is a processing pipeline of three steps: **(1)** `Recv:` Receive model updates from all assigned producers. The received model update is enqueued in a FIFO queue. In LIFL, the object key of the model update is enqueued as the actual model update resides in shared memory; **(2)** `Agg:` Aggregator dequeues a model update from the FIFO queue in `Recv` and then aggregates it. The `Agg` step checks if the aggregation goal is met after the dequeued update is aggregated. If the aggregation goal is not met, `Agg` is repeated until the aggregation goal is met, before moving to `Send`; and **(3)** `Send:` sends the final model update to the designated consumer. The execution of `Recv` and `Agg` overlaps to enable eager aggregation, *i.e.,* once the `Recv` step receives a model update, it immediately passes the model update to `Agg` step for aggregation.

6.5 LIFL's Control Plane Design

6.5.1 Locality-aware Placement and Load Balancing

The placement of aggregators can lead to different routing behaviors: When aggregators with cross-level data dependencies are placed on the same node, the shared memory

processing and eBPF-based sidecar can facilitate intra-node routing. When these aggregators are placed across different nodes, the gateway has to perform inter-node routing. To minimize the transfer of model updates in LIFL, we take a data-centric strategy like [223] that is aware of the locality of model updates and places the aggregator close to the model updates. As such, the in-place message queuing (§6.4.2), which is, in fact, the result of load balancing (clients to worker node mapping), directly affects the effectiveness of the locality-aware placement of the aggregators.

Our objective of load balancing involves two crucial criteria: **(1)** Minimizing inter-node communication while maximizing the utilization of shared memory within each node. **(2)** Ensuring the residual service capacity of the worker node meets the demand; the residual service capacity ($RC_{i,t}$) of worker node i at time t is determined by $RC_{i,t} = MC_i - (k_{i,t} \times E_{i,t})$. Here, MC_i represents the maximum service capacity, denoting the maximum number of model updates that can be aggregated simultaneously on worker node i. The value of $k_{i,t}$ is the arrival rate of model updates directed to worker node i at time t, and $E_{i,t}$ is the average execution time required to aggregate a model update on node i. We can also get a coarse-grained estimate on the queue length ($Q_{i,t} = k_{i,t} \times E_{i,t}$) of node i at time t.

LIFL actively monitors both $E_{i,t}$ and the arrival rate $k_{i,t}$ using the sidecar in §6.4.3. We determine the value of MC_i offline. We incrementally increase the arrival rate k_i to node i. Let k'_i and E'_t denote the arrival rate and average execution time at the point we observe a significant increase in E_i. This indicates that node i is becoming overloaded and we estimate MC_i as $k'_i \times E'_i$.

We approach the load balancing task as a bin-packing problem, aiming to allocate model updates from clients to a minimal number of worker nodes, while ensuring that the residual service capacity of each worker node is not exceeded. This naturally reduces the inter-node communication as much as possible, since the communication between a particular pair of worker nodes only happens once. We use *BestFit* for the bin-packing, as it concentrates load onto the fewest nodes possible, to reduce inter-node traffic and maximize shared memory use. In contrast, *WorstFit* spreads the load across more nodes, similar to the "Least Connection" policy in Knative (§6.6.1). Furthermore, *FirstFit* focuses on reducing search complexity without being locality-aware.

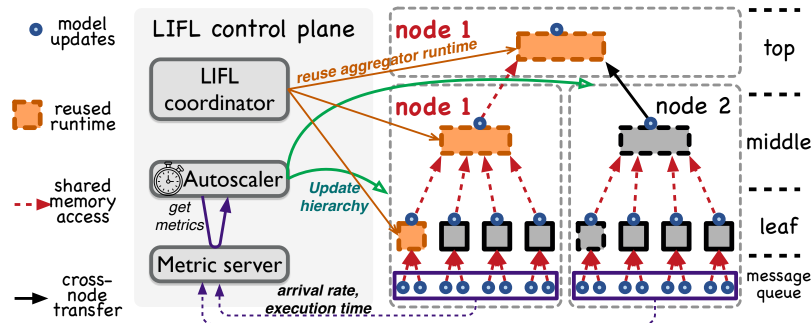


Figure 6.8: Control plane orchestration in LIFL: The autoscaler periodically re-plans the hierarchy based on the arrival rate of each worker node. The LIFL coordinator applies reusing of aggregators.

6.5.2 Planning the Hierarchy for Aggregation

The goal of hierarchy-aware autoscaling is to maximize the parallelism of aggregation at each level, given the number of model updates to be aggregated. This can minimize the completion time of each level and thus minimize the aggregation completion time (ACT)

for hierarchical aggregation. We plan a hierarchical aggregation structure within each node, tailored to the number of pending model updates ($Q_{i,t}$) in the message queue. Every node produces an intermediate model update that is dispatched to the node chosen to have the top aggregator that updates the global model. This approach significantly reduces the need for cross-node transfers for intermediate model updates.

LIFL periodically adjusts (*i.e.,* scales) the hierarchy on node i, guided by our estimates of $Q_{i,t}$. To prevent excess resource allocation due to short-term spikes in $Q_{i,t}$, we employ the Exponentially Weighted Moving Average (EWMA) to smooth $Q_{i,t}$: $Q_{i,t} = \alpha \times Q_{i,t-1} + (1 - \alpha) \times Q_{i,t}$, where α is the EWMA coefficient. We set $\alpha = 0.7$ based on it yielding the best results in our experiments. Our current implementation supports a two-level k-ary tree hierarchy on each node, comprising a "central" middle aggregator responsible for aggregating model updates from $Q_{i,t}/I$ leaf aggregators, where I is the number of model updates of clients per leaf aggregator. Given that the steps within a LIFL aggregator (Fig. 6.7) are executed sequentially, we want to maximize the parallelism by having a limited I to be small (*e.g.,* at 2), ensuring that a leaf aggregator experiences minimal waiting time after receiving the initial update from the first client.

LIFL re-plans the hierarchy on each worker node periodically. This involves estimating $Q_{i,t}$ across the worker nodes and creates/terminates aggregators accordingly. The LIFL control plane updates the routes between aggregators based on the renewed hierarchy (details in §6.4.4).

6.5.3 Opportunistic Reuse of Aggregator Instances

The scaling policy in LIFL incorporates an opportunistic "reuse" scheme to maximize the utilization of warm aggregator instances since aggregators in LIFL use homogenized runtimes (Fig. 6.7) with the same code and libs. This sidesteps the cascading effect [173] when starting up a hierarchy of aggregators (in fact function chains).

Given a hierarchy of aggregators selected on the node, LIFL picks a leaf aggregator that has already completed its aggregation task and is idle. LIFL converts its role to a middle aggregator on that node. No further change is required as LIFL's aggregator runtime is stateless. LIFL selects the first middle aggregator that completes its local aggregation task and converts it to be the top aggregator responsible for updating the global model. This minimizes the need to start up new instances for higher-level aggregators, and avoids additional startup delays.

6.5.4 Eager aggregation in LIFL

LIFL employs eager aggregation (Fig. 6.1) leveraging its more flexible and dynamic timing of the aggregation process. Eager aggregation performs timely aggregation as model updates arrive, even if it triggers the cold start of an aggregator (when no idle-but-warm aggregator is available). This takes advantage of the *overlap* between the start-up delay and transfers of model updates, allowing eager aggregation to mask cold starts up until the last model update. It also mitigates congestion that can occur when trying to aggregate all model updates simultaneously. In contrast, lazy aggregation aggregates all model updates in a batch when the aggregation goal is reached. But, the arrival of local model updates from trainers can be spread over a relatively long duration. Our evaluation shows eager

aggregation achieves a 20% reduction on ACT (Fig. 6.11(a)). We implement eager aggregation in LIFL following the step-based processing model described in §6.4.6. LIFL updates the version of the global model whenever the aggregation goal is achieved.

6.6 Evaluation & Analysis

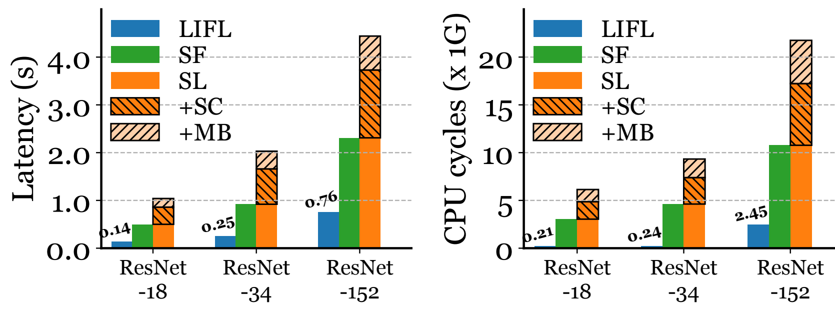
We quantify the performance gain and resource savings by using LIFL, starting with analyzing a set of microbenchmarks to understand the different design considerations of LIFL, including shared memory processing, the effectiveness and overheads of LIFL's orchestration scheme. We then demonstrate the benefits of LIFL from a system-level perspective using real FL workloads.

Baseline Systems: We implement several baseline FL systems for LIFL to compare against. **(1) "Serverful system" (SF):** The "serverful system" is implemented following the design described in [87] and [126]. Both of them adopt the architecture depicted in Fig. 6.2 (a). **(2) "Serverless system" (SL):** The baseline "serverless system" is implemented following the design described in FedKeeper [94] and AdaFed [133] that uses the architecture depicted in Fig. 6.2 (b). We choose Knative [59] as the serverless framework to build these alternatives. We utilize the open-source Flame platform [57] to provide necessary FL components, *e.g.,* coordinator, selector, aggregator, and client.

Implementation of LIFL: We implement LIFL based on SPRIGHT [181], a lightweight, high-performance serverless framework. LIFL includes object store support, model checkpoints, and routing support for hierarchical aggregation. LIFL uses Python's multiprocessing package to implement the shared memory pool instead of the DPDK-based shared

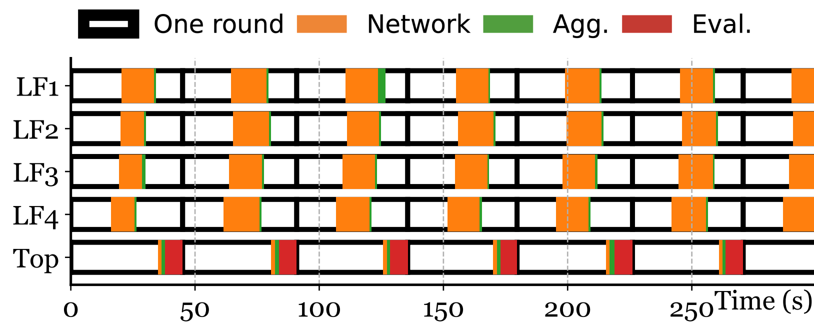
memory pool used in the original implementation of SPRIGHT. The current implementation of LIFL only supports synchronous FL. Supporting asynchronous FL is part of our future work.

Testbed setup: We leverage the NSF Cloudlab [102]. The nodes we used have a 64-core Intel Cascade Lake CPU@2.8 GHz, 192GB memory, and a 10Gb NIC. We use Ubuntu 20.04 with kernel version 5.16.



(a) latency of a single model update transfer (intra-node)

(b) CPU usage of a single model update transfer (intra-node)



(c) LIFL's Aggregation Timing (with ResNet-152)

Figure 6.9: Data plane improvement for hierarchical aggregation: Serverful (`SF`), Serverless (`SL`), and LIFL. `SL`'s latency includes contributions of `+SC` (sidecar) and `+MB` (message broker).

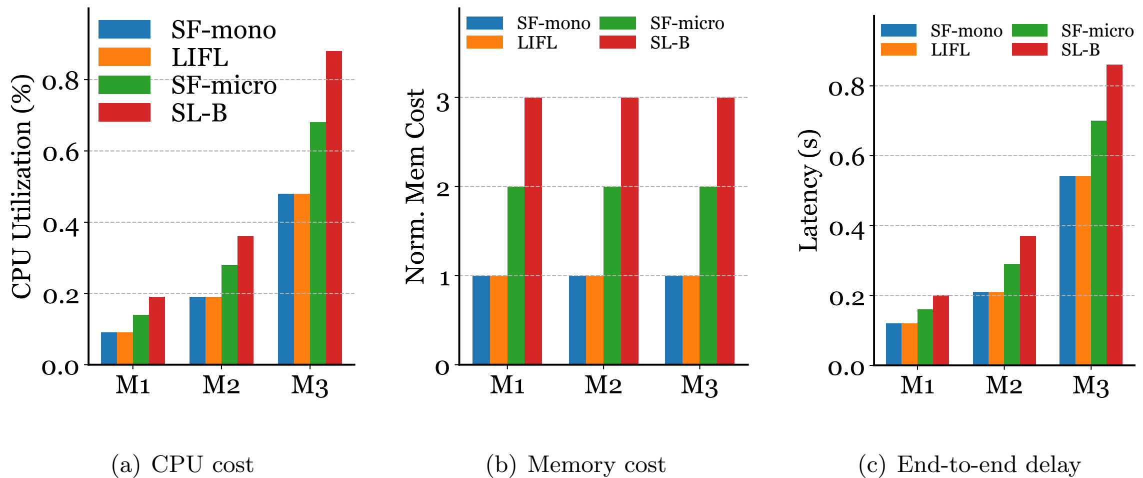
6.6.1 Microbenchmark Analysis

Data plane improvement for hierarchical aggregation: To understand the improvements in data plane performance of hierarchical aggregation with LIFL's shared memory processing, we use the same aggregation hierarchy as in §6.4.1, comprising one top aggregator and four leaf aggregators. All aggregators are placed on the same node.

We consider the following serverful and serverless alternatives: (1) The serverful setup (`SF`) establishes direct networking channels (based on gRPC) between leaf aggregators and the top aggregator; (2) the serverless setup (`SL`) uses indirect networking to connect leaf aggregators and the top aggregator, through a message broker on the same node. Each aggregator has a container-based sidecar to mediate inbound and outbound traffic; (3) the LIFL setup uses shared memory for communication between aggregators. We consider three ML models with distinct sizes: ResNet-18 (∼44MB), ResNet-34 (∼83MB), and ResNet-152 (∼232MB).

Fig. 6.9(a) shows the latency breakdown of a single model update transfer between the leaf aggregator and top aggregator for different model sizes. We specially mark the share of sidecar (`+SC`) and message broker (`+MB`) for the serverless setup. `SL` consistently results in 2× and 6× higher latency than `SF` and LIFL, respectively. The significant CPU usage of `SL` (Fig. 6.9(b)) clearly shows the poor efficiency and performance of the indirect networking used in the serverless setup, caused by its use of the message broker and heavyweight sidecar. We see that LIFL is considerably better than `SF` and `SL` in terms of both CPU usage and latency.

Fig. 6.9(c) shows the timing of various FL processing tasks during hierarchical aggregation when using LIFL's data plane. It is clear that LIFL's shared memory processing helps reduce the overhead and improve the performance of the data plane with hierarchical aggregation. LIFL completes each round in just 44.9 seconds compared to 57 seconds on average even for the serverful setup in Fig. 6.4. Further, through careful placement, aggregators in LIFL can fully exploit the high-speed intra-node data plane over shared memory, as discussed next.



(a) CPU cost (b) Memory cost (c) End-to-end delay

Figure 6.10: Message queuing overheads.

In-place message queueing benefit We examine LIFL's in-place message queuing through a comparison with the serverful and serverless alternatives depicted in Fig. 6.5, including the *monolithic* serverful setup (denoted as `SF-mono`), the *microservice*-based serverful setup (denoted as `SF-micro`), and the *basic* serverless setup (denoted as `SL-B`). We quantify the overheads of message queuing for a single model update transfer between the client to the aggregator. We consider three metrics: (1) the total memory consumed for queuing the

model update along the data pipeline; (2) the CPU cycles spent in the data pipeline; and (3) the end-to-end networking delay from the client to the aggregator. Note that we exclude the overhead on the client-side. We consider three ML models with distinct sizes: (`M1`) ResNet-18 (∼44MB), (`M2`) ResNet-34 (∼83MB), and (`M3`) ResNet-152 (∼232MB).

Fig. 6.10 shows the results of CPU, memory cost and end-to-end networking delay. The memory consumption in `SF-mono` is mainly from the in-memory queue inside the aggregator. For LIFL it is primarily consumed by the shared memory used to buffer the model update. But, `SL-B` consumes 3× more memory than `SF-mono` and LIFL. The extra memory consumption of `SL-B` comes from the use of sidecar and message broker, both of which need to locally buffer the model update. `SF-micro`, on the other hand, saves one queuing stage at the sidecar, but still incurs the queuing at the message broker and consuming extra memory. LIFL's in-place message queuing totally eliminates these unnecessary queuing stages.

Looking at the CPU consumption, LIFL is ∼1.5× and ∼1.9× less than `SL-B` and `SF-micro`, respectively. In terms of the end-to-end networking delay (client to aggregator), LIFL is ∼1.3× and ∼1.7× less than `SL-B` and `SF-micro`, respectively. LIFL's improvement in CPU cost and networking delay, compared to `SL-B` and `SF-micro`, are also a result of the elimination of the sidecar and message broker from the data pipeline, and the message queuing if far more efficient. This illustrates the benefits of LIFL's in-place message queuing, achieving the equivalent efficiency and performance of a monolithic, serverful design (with far less resource consumption as we see for typical FL applications).

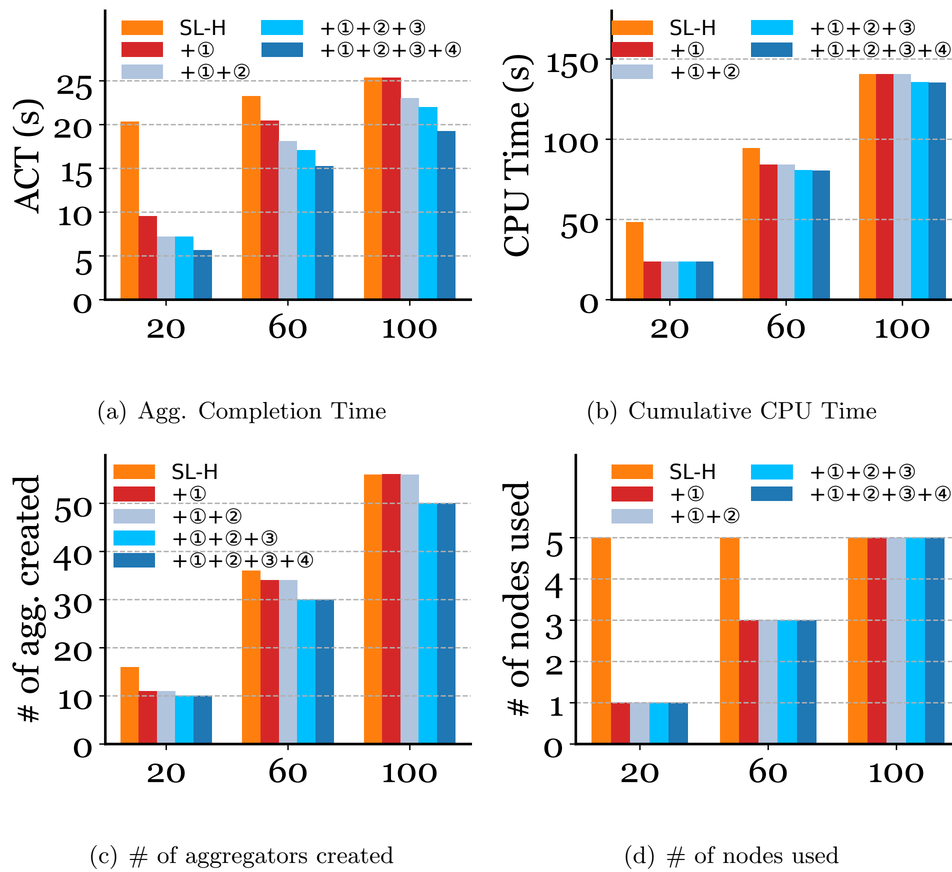
Stateful "tax" in LIFL The per-node gateway is a key component that enables a number

of data plane functionalities in LIFL, including in-place message queuing and inter-node data transfer. Unlike stateless aggregators, the gateway is deployed as a stateful, persistent component on every LIFL worker node. This raises the concern about the stateful "tax", *i.e.,* the CPU/memory cost of having stateful components in the FL system.

On the other hand, a stateful "tax" of some form commonly exists in serverful and serverless alternatives, as shown in Fig. 6.5. The stateful component in a monolithic serverful setup is the aggregator itself, running as an "always-on" monolith. In the microservice-based serverful setup, the message broker is the stateful component, as is the case for the basic serverless setup. We quantitatively compare the stateful "tax" of LIFL's gateway with serverful and serverless alternatives in Fig. 6.5. The result in §6.4.2 shows that stateful "tax" in LIFL is the lowest.

Improved orchestration in LIFL: We now quantify the benefits of LIFL's orchestration in improving hierarchical aggregation. We demonstrate the effectiveness of LIFL by applying: ① locality-aware placement (§6.5.1), ② hierarchy-planning (§6.5.2), ③ aggregator reuse (§6.5.3), and ④ eager aggregation (§6.5.4) step-by-step. We use five nodes for this experiment. The maximum service capacity (MC_i) of each node in our testbed is 20.[5] We focus on two aspects: resource consumption and Aggregation Completion Time (ACT) to aggregate a given number of model updates. In this experiment, we assume the estimated $Q_{i,t}$ is equal to the actual queue length on each active node. We focus on the importance of having warm aggregators based on the pre-planned hierarchy, to avoid the cold start penalty.

[5]Our testbed nodes are homogeneous, hence all MC_i are the same. With heterogeneous nodes, MC_i may vary.



(a) Agg. Completion Time

(b) Cumulative CPU Time

(c) # of aggregators created

(d) # of nodes used

Figure 6.11: Improvement with LIFL's orchestration, with ⓘ being additions to baseline LIFL; *x-axis is the number of model updates arriving at the aggregation service concurrently.*

We compare LIFL against a baseline serverless control plane using hierarchical aggregation (`SL-H` in Fig. 6.11). `SL-H` employs LIFL's shared memory data plane (so both have the same data plane) with Knative's "Least Connection" load balancing strategy [166] that assigns newly arrived model updates to the node with the smallest queue length. The aggregators in `SL-H` use lazy aggregation by default. The ML model used is ResNet-152. Note that the latency to transmit a single model update of ResNet-152 across nodes (on the current testbed) is ∼4.2 seconds.

By using locality-aware placement, LIFL also achieves 2.1× and 1.13× ACT reduction than `SL-H` (for 20 and 60 model updates in Fig. 6.11(a)). This improvement is attributed to LIFL's bin-packing strategy, which effectively consolidates aggregators onto the same node to fully exploit shared memory processing. Applying hierarchy-planning and reusing warm aggregator instances (+①+②+③) further reduce ∼1.22× ACT of LIFL, as keeping aggregators warm mitigates the cold start delay that exists in both `SL-H` and (+①). Further, after enabling eager aggregation (+①+②+③+④), LIFL allows higher-level aggregators to consume and aggregate the model updates in a timely manner, effectively avoiding the intermediate model updates (produced by the lower-level aggregators) being queued up at the higher-level aggregators. This saves ∼1.2× in ACT compared to (+①+②+③) that uses lazy aggregation.

While being effective in reducing ACT, LIFL also helps to reduce costs. Just using locality-aware placement (+① in Fig. 6.11(b)), LIFL can save considerable CPU overhead by reducing inter-node data transfers (with 20 and 60 model updates). Enabling aggregator reuse saves additional CPU cycles, as it avoids having the CPU initialize new

aggregators. For 100 model updates though, the service capacity of all five nodes would be maxed out, reaching the limit of the benefit of LIFL's orchestration. However, the data plane improvement of LIFL can still make it outperform the basic serverful and serverless setups, as demonstrated in Fig. 6.9.

As shown in Fig. 6.11(c), LIFL reduces the number of aggregators created, by packing more aggregators into fewer nodes. After we apply locality-aware placement to LIFL (+①), LIFL can also reduce the number of nodes used considerably (see Fig. 6.11(d)): Given 20, 60, and 100 model updates, LIFL's locality-aware placement efficiently packs them into 1, 3, and 5 nodes, respectively. This avoids repeatedly creating a middle aggregator on each of the 5 nodes (except when the service capacity of all 5 nodes is fully consumed). On the other hand, `SL-H` uses all 5 nodes throughout, uniformly distributing model updates across all 5 available nodes. This will lead to additional cross-node data transfers, regardless of available model updates. Note that the service capacity of all 5 nodes is fully consumed for 100 model updates.

Orchestration overhead of LIFL: We evaluate the orchestration overhead of LIFL, given a different number of clients. The time for completing the locality-aware placement in LIFL is less than 17 milliseconds, even with 10K clients, which is the maximum number of client settings observed in Google's production FL stack [87]. Compared to the ACT, which takes several tens of seconds with a large amount of clients, this overhead for locality-aware placement is negligible. The EWMA estimator for hierarchy-planning takes 0.2 milliseconds per estimate, which is also negligible compared to the 2-minute cycle time used by LIFL to re-plan the hierarchy on each worker node. The aggregator reuse and eager aggregation

incur almost no overhead, as they do not require active involvement of the LIFL control plane.

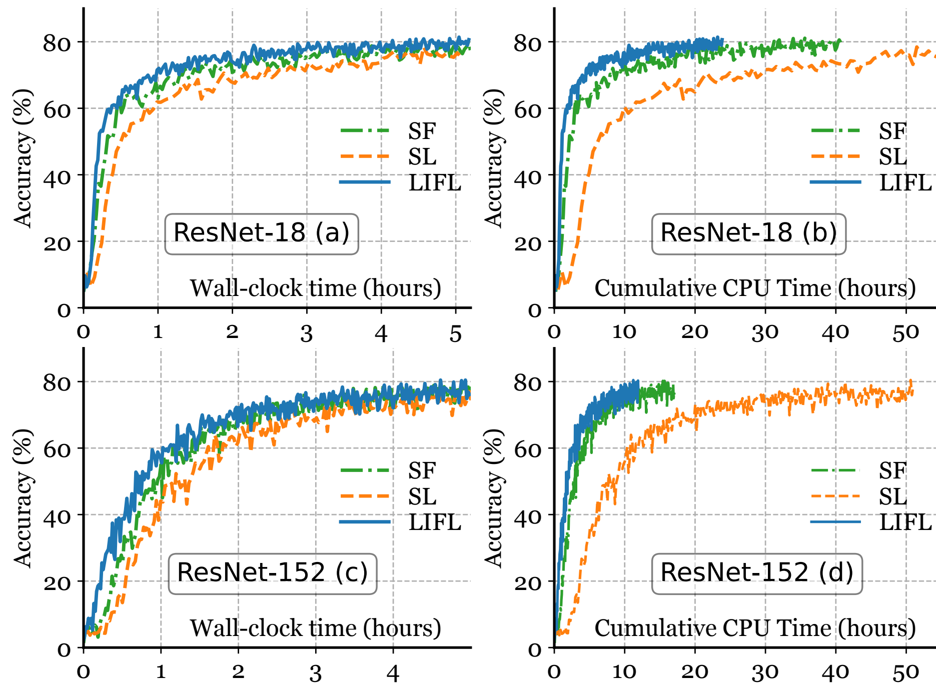
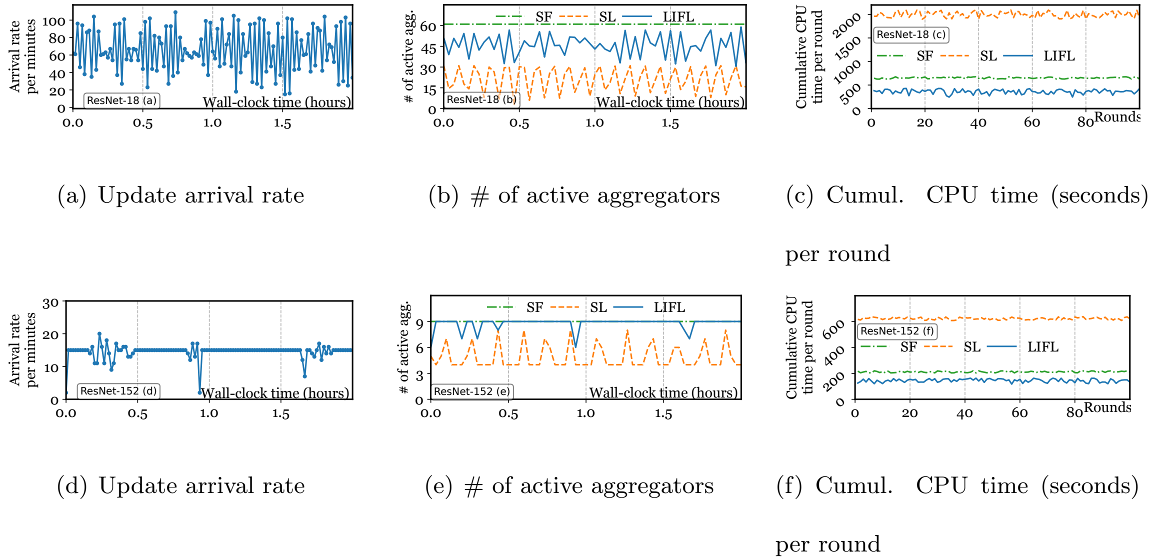


Figure 6.12: **ResNet-18:** (a) Time-to-accuracy and (b) Cost-to-accuracy; **ResNet-152:** (c) Time-to-accuracy and (d) Cost-to-accuracy.

6.6.2 FL Workloads Setup

Our aim is to demonstrate the generality of LIFL in improving performance and reducing the cost of FL from a system-level perspective. We consider synchronous FL (using FedAvg [162]) to justify LIFL's design. We use Stochastic Gradient Descent on the client. Clients are configured with a batch size of 32 in a local training epoch, with the learning rate set to 0.01.



(a) Update arrival rate

(b) # of active aggregators

(c) Cumul. CPU time (seconds) per round

(d) Update arrival rate

(e) # of active aggregators

(f) Cumul. CPU time (seconds) per round

Figure 6.13: **ResNet-18 (a, b, c), ResNet-152 (d, e, f):** Time series of arrival rate, number of active aggregators, and cumulative CPU time (seconds) per round.

Benchmark selection: We consider image classification, training ResNet [124] models with the FEMNIST dataset [221]. We use non-IID datasets from FedScale [145] (with its real client-data mapping) to keep the setting realistic with different data distributions across the client population.

Configuration of clients: We consider two distinct client setups: (***ResNet-18*** setup) We use the client in this setup to train a ResNet-18 model. Clients are considered to be mobile devices with limited computing capacity, available only when each has battery power and is connected to a data (*e.g.,* WiFi) network. This results in high variability in the number of mobile devices available to perform training tasks. As such, we let each client hibernate for a random interval within [0, 60] seconds to emulate the dynamic availability of typical mobile device behavior. This generates varying loads over time, as shown in Fig. 6.13(a), justifying the need for scaling with a serverless framework as well as LIFL.

(***ResNet-152*** setup) The client in this setup trains the relatively heavyweight ResNet-152 model. The client is considered to be a server with substantial computing capacity and is highly available. As such, we keep clients in this setup always-on. This results in a more stable arrival pattern of model updates, as shown in Fig. 6.13(d).

We use a total of 20 physical nodes with 5 nodes used to run aggregators. We use 4 nodes as leaf/middle aggregators and dedicate one node to be the top aggregator. To deliver the benefits of a "serverful system" (`SF`), we always maximize the resource allocation to the aggregators and keep them warm throughout the experiment. For the serverless setup (`SL` and LIFL), we create aggregators on demand.

We use the remaining 15 physical nodes to run the clients. In the ***ResNet-18*** setup, since we consider clients to be compute-constrained mobile devices, we run eight clients on the same physical node, so each client only gets a small share of the compute capacity of the physical node. Therefore, in the ***ResNet-18*** setup, we can keep 120 simultaneously active clients in each round. In the ***ResNet-152*** setup, we treat the client as a server node, so we dedicate a physical node for a ResNet-152 client. In this ***ResNet-152*** setup, we keep 15 simultaneously active clients in each round. The active clients are selected from a total of 2,800 real clients provided by FedScale [145].

6.6.3 Putting It All Together

(***ResNet-18***) **Time to Accuracy:** We compare the time-to-accuracy of LIFL against `SL` and `SF`. To reach 70% accuracy of ResNet-18 (Fig. 6.12 (a)), LIFL takes only 0.9 hours (wall clock time), which is 1.6× faster than `SF` (1.4 hours). Compared to `SL` which takes 2.4 hours, LIFL is 2.7× faster. The improvement with LIFL can be attributed to the shared

memory data plane and the improved orchestration to effectively utilize resources, thereby reducing ACT (see §6.6.1).

The time spent by the `SL` aggregation service increases due to a combination of factors including sidecar overhead, function chaining, and simplistic orchestration. Frequent start-up of the aggregators in `SL` (Fig. 6.13(b)) adds delays to the aggregation (for the first arrival update in a round). This increased aggregation time of `SL` eventually hurts the time-to-accuracy (70%), making it even slower than `SF`.

(***ResNet-18***) **Cost savings with LIFL:** LIFL achieves significant cost savings compared to `SF` and `SL`. We focus on the cumulative costs (CPU time) consumed by the aggregation service to achieve a certain model accuracy. To reach the 70% accuracy level of ResNet-18 (Fig. 6.12 (b)), LIFL consumes 4.5 CPU hours, which is 1.8× less than `SF` (8 CPU hours). Further, `SF`, with its simplistic fixed resource allocation, keeps aggregators "always-on", constantly occupying its CPU allocation (Fig. 6.13(b)). LIFL adapts well to the arrival rate of model updates and re-plans (scales) the hierarchy accordingly, using resources to match demand. Also note that the LIFL's aggregator, when deployed as a Kubernetes pod or container, is also cheaper (smaller resource allocation) than `SF`, as LIFL requires less CPU to complete the same amount of aggregation tasks (Fig. 6.13(c)).

In contrast, `SL` consumes much more CPU (26 CPU hours) to achieve the 70% accuracy level of ResNet-18 (Fig. 6.12 (b)) compared to LIFL (4.5 CPU hours). Although `SL` has relatively fewer active aggregators over time (Fig. 6.13(b)), its data plane and sidecar overheads, and the CPU consumed for start-up results in `SL` having more than 5× the CPU consumption of LIFL. This higher CPU time cost per round (for the same amount

of aggregation work completed) requires the cloud service provider to allocate far more resources to the aggregator (*e.g.,* as a pod), making a single aggregator in `SL` much more expensive than both `SF` and LIFL.

(***ResNet-152***) **Time to Accuracy:** Fig. 6.12 (c) shows the time-to-accuracy of the different alternatives for RestNet-152. To reach 70% accuracy, LIFL takes 1.9 hours (wall clock time), which is 1.15× faster than `SF` (2.2 hours). Comparatively, `SL` takes 3.2 hours. LIFL is 1.68× faster than `SL`. The heavy-weight sidecar, slow function chaining, function startup delays, and simplistic orchestration, are responsible for the larger time-to-accuracy of `SL` for ResNet-152, just as we saw with the ResNet-18 workload, as well as with the microbenchmark analysis.

(***ResNet-152***) **Cost savings with LIFL:** As Fig. 6.12 (d) shows, LIFL again achieves significant cost savings (on cumulative CPU time) compared to `SF` and `SL`. To reach the 70% accuracy level of the ResNet-152 model, LIFL consumes 4.76 CPU hours, which is 1.43× less than `SF` (6.81 CPU hours). In contrast, `SL` consumes much more CPU (20.4 CPU hours) to achieve the same 70% accuracy level compared to LIFL. This again is consistent with what we observed from the ResNet-18 workload, highlighting the advantage of LIFL.

Summary: LIFL takes advantage of the fine-grained elasticity of serverless to scale the aggregation service based on load changes, saving CPU consumption compared to serverful alternatives. When comparing LIFL with `SL`, LIFL is even more compelling, with far lower CPU consumption because of LIFL's orchestration scheme and lightweight data plane (as we saw from the microbenchmark analysis). Thus, LIFL shows that it truly leverages the elasticity promise of the serverless computing paradigm.

6.7 Conclusion

LIFL is an optimized serverless FL system aimed at making FL more efficient and significantly lowering its operational cost. LIFL adopts hierarchical aggregation to support FL at scale. Its serverless infrastructure leverages shared memory processing to offer high-speed yet efficient intra-node data plane and event-driven sidecar functionality to facilitate communication within hierarchical aggregation. LIFL's orchestration scheme adjusts the aggregation hierarchy based on load and, maximizes the utilization of shared memory through intelligent placement and reuse of aggregation function instances, thus saving the cost. Our evaluation shows that LIFL's optimized data and control planes improve the resource efficiency of the aggregation service by more than **5**×, compared to existing serverless FL systems, with **2.7**× reduction on time-to-accuracy for ResNet-18. LIFL also achieves **1.8**× better efficiency and **1.6**× speedup on time-to-accuracy than a serverful system. In training ResNet-152 to reach 70% accuracy, LIFL is **1.68**× faster than an existing serverless FL system, while reducing CPU costs by **4.23**×.

Chapter 7

High-performance, 3GPP-compliant 5G and Beyond Systems

7.1 Introduction

The demand for 5G and beyond technologies is being driven by the emergence of applications like the Internet of Things (IoT) and connected vehicles, which rely heavily on cellular networks for ubiquitous access and low latency. Further, the deployment of 5G, especially the 5GC and (soon) the Radio Access Network (RAN), in cloud infrastructure has been instrumental in its widespread implementation as well as its scalability. Cloud-based 5G core networks allow for efficient resource provisioning, using seamless scaling to accommodate the diverse demands of connected User Equipment (UE) and applications.

The disaggregated 5GC architecture represents a transformative approach to implementing 5G core networks, moving away from the monolithic, tightly integrated network elements of previous approaches to build the cellular core to a flexible and scalable approach based on microservices. The various components of the 5GC are implemented as software-based Network Functions (NFs), interconnected as a chain to accomplish the required functionality. Each NF is implemented as an individual microservice, focusing on a specific task, such as Access and Mobility Management, Session Management, Authentication, *etc.* The use of microservices enables fine-grained control and allows for rapid deployment and upgrades of individual components without affecting the entire 5G system. Additionally, this disaggregated approach enables resource optimization, since NFs can be dynamically scaled based on traffic demand.

For a seamless end-to-end low-latency user experience, both the radio access and as well as the core components of 5G cellular networks have to improve. Advancements in radio access technology, such as millimeter wave, have reduced access network latency to approximately the order of a few milliseconds (possibly 1 ms [176]). The recent effort shows that electronic mmWave beam alignment and link acquisition can be completed within 1-10 ms, allowing a UE's connection establishment with the gNodeB to be completed quickly [122]. In addition, the advent of disaggregated 5GC has spurred significant efforts to re-architect the data plane to meet the stringent requirements of performance and scalability. A variety of optimizations have been explored to enhance the 5GC data plane, including DPDK [130], eBPF [175], SmartNIC [172], and offloading to hardware switches using P4 [161].

Furthermore, to address the challenge of achieving low-latency communication in the 5GC control plane, efforts have been made to explore innovative approaches that harness high-performance shared memory I/O (*i.e.,* data exchange) for information exchange between NFs implementing microservices in the 5GC. By leveraging shared memory processing, the 5G control plane can significantly reduce the delays caused by data copies and protocol processing, resulting in improved response times and a more seamless user experience. In [130], we discussed $L^2$5GC, a state-of-the-art 5G control plane design developed on top of OpenNetVM [226], a high-performance shared-memory NFV platform. $L^2$5GC utilizes shared memory processing among the 5G control plane components, which reduces the completion time of control plane events (*e.g.,* UE registration, handover, paging) by almost 50% on average [130].

Despite architectural advancements, limitations persist in both the control and data planes of the 5GC design in achieving the high performance goals:

1. The *control plane* still contributes substantially to the overall high latency observed in the 5GC. One major contributor is the potential for increased mobility handovers, driven by the wide adoption of millimeter-wave cells [216], which have smaller cell sizes as well as limited coverage, leading to more frequent handover events. These handovers have to be handled by the 5G control plane. Additionally, with the need to conserve energy in batteries on UEs like mobile phones as well as IoT devices, there will likely be much more idle-active transitions among the UEs. The proliferation of 5G UEs (*e.g.,* mobile phones, IoT devices, autonomous vehicles) further increases the load on the 5G control plane. The completion times of control plane events, for

instance, a handover process taking 1.9 seconds [154], directly influence the delay and packet loss encountered by the data packets transmitted to an end-user device.

2. A crucial implementation feature of the 5G control plane is the Service-based Interface (SBI) recommended by 3GPP, which has been the de-facto communication standard used for communication between disaggregated 5G control plane NFs, relying on HTTP/REST-based communication. However, the use of SBI introduces a number of overheads, such as data copies, protocol processing, and user-kernel space boundary crossings [130, 131, 182]. These overheads can result in increased latency, apart from the penalty due to the traditional cellular control plane core procedures (details in §3.4).

3. The data plane faces challenges to efficiently perform packet classification, particularly for emerging cellular use cases involving an increasing number of packet detection rules (PDRs).

Although, the state-of-the-art 5GC design, $L^2$5GC [130], delivers significant performance improvement in the 5GC control plane, $L^2$5GC's control plane only supports a limited number of user sessions due to a rather limited implementation of the shared memory I/O to replace the SBI.[1] $L^2$5GC chose to use raw shared memory I/O provided by DPDK, which operates *asynchronously* between caller (source) and callee (destination). For asynchronous data exchange, the caller typically continues with other tasks without being blocked and does not wait for a response from the callee. However, this is incompatible with the HTTP/REST-based SBI, which primarily operates *synchronously* between

[1] Based on examining the source code [45] of $L^2$5GC at the latest commit hash `74cb035`.

caller and callee, *i.e.,* the caller sends a request to the callee and waits until a response is returned. The mismatch between L^{2}5GC's asynchronous shared memory I/O and synchronous SBI makes it hard to harmonize them, unfortunately increasing the complexity of code development and the difficulty of code maintenance and updates of L^{2}5GC. Further, L^{2}5GC's shared memory I/O only supports stateless processing. This lack of capability to preserve connection context makes L^{2}5GC's shared memory I/O connection-agnostic and not able to distinguish between different user sessions. The implementation complexity and the statelessness of L^{2}5GC's shared memory I/O eventually impede L^{2}5GC's ability to scale up, supporting multiple user sessions.

In addition to the mismatch between synchronous and asynchronous I/O, another challenge comes from programming language incompatibility. L^{2}5GC is adapted from our earlier work on a 3GPP-compliant 5GC implementation, free5GC [58]. For the purpose of functionality and development velocity, free5GC chose to use Golang, a high-level programming language, in its implementation. On the other hand, L^{2}5GC's asynchronous shared memory I/O is developed with the C-based DPDK libraries for high-performance networking. This leads to a need for substantial re-factoring of code when porting the 3GPP-compliant free5GC to L^{2}5GC to reduce the control plane latency.

As we look ahead to the potential of 5G wireless environments, it is imperative to address the above limitations. We thoroughly examine these constraints and propose `L`^{2}`5GC+` [159, 182], an enhancement to L^{2}5GC. `L`^{2}`5GC+` takes advantage of our newly designed shared memory I/O interface, X-IO [184], and tackles the pain points of L^{2}5GC we have outlined above, including limited user session support and the need for complex code

refactoring when porting free5GC's (or other traditional SBI-based) control plane implementation, while retaining the performance benefits of L^{2}5GC's shared memory processing. To achieve this, we re-design the shared-memory-based networking stack in L^{2}5GC to support *synchronous* I/O between control plane NFs. This avoids heavyweight kernel-based networking used in HTTP/REST-based SBI, while being strictly 3GPP-compliant. To support multiple user sessions simultaneously in shared memory processing, L^{2}5GC+ introduces necessary connection establishment and teardown procedures. Important connection states, such as caller and callee ID[2] are kept in a state map maintained in L^{2}5GC+'s shared memory networking stack. This enables L^{2}5GC+'s shared memory I/O to be aware of distinct connections, on top of which L^{2}5GC+ can distinguish different user sessions, unlike L^{2}5GC. Our contributions encompass:

- *Enhancing PDR Rule Lookups:* We explore innovative packet classification mechanisms to expedite PDR lookups and update operations in the UPF, going beyond the linear search approach recommended by 3GPP.

- *Evolving to a High-performance SBI:* L^{2}5GC+ utilizes shared memory processing to offer a low-latency data path for exchanging control plane messages. L^{2}5GC+ adds ***synchronous*** I/O support on top of ***asynchronous*** shared memory I/O designed primarily for Layer-2 NFs, ensuring our high-performance SBI is still compliant with 3GPP standards.

- *Being strictly 3GPP-compliant:* To speed up the development velocity when porting free5GC to L^{2}5GC+, we expose the equivalent SBI APIs from L^{2}5GC+'s networking

[2]Similar to kernel-based TCP/IP stack, L^{2}5GC+ uses IP and port numbers to differentiate between NFs using shared memory communication.

stack. By leveraging the cross-language support offered by the CGo interface [49], we mitigate the programming language incompatibility between the lower-layer shared memory transport (developed with C-based DPDK libraries) and upper-layer Golang-based SBI APIs. This allows us to seamlessly replace the kernel-based SBI APIs for existing free5GC control plane NFs, while keeping the NF implementation *unchanged*, thus greatly reducing porting efforts and being 3GPP-compliant.

To gain a solid understanding of how $\texttt{L}^2\texttt{5GC+}$ actually performs, we evaluate $\texttt{L}^2\texttt{5GC+}$ on a commercial testbed with an increasing number of UEs. We select several representative control plane events, including UE registration, PDU session establishment, to evaluate $\texttt{L}^2\texttt{5GC+}$ against a popular 5GC implementation, free5GC [58], which uses kernel-based SBI in the control plane. Results demonstrate the performance improvement of $\texttt{L}^2\texttt{5GC+}$'s shared memory SBI, especially when there are multiple user sessions operating concurrently in the 5GC control plane. We have an open-source 5GC implementation that incorporates some of these proposed $\texttt{L}^2\texttt{5GC+}$ components at `https://github.com/nycu-ucr/L25GC-plus.git`

7.2 Background and Motivation

7.2.1 Evolving the Architecture of the Cellular Core Network

The cellular core network forms the heart of an operator's cellular network. A cellular network's packet processing functions are typically divided between the RAN and the core network. Figure 7.1 exemplifies the typical architecture of the 5GC adopted by implementations of the 3GPP standard [130]. The RAN manages wireless channel resources and processing at the cellular base stations, to connect UEs (*i.e.,* typically mobile client

devices) and the cellular core. A 5GC is further split into the data/user and control planes. The User Plane Function (UPF) is a key NF in the 5GC data plane, which connects UEs to the Data Network (DN) to access data (e.g., Internet) services.

It is useful to understand the evolution of the core across generations of the cellular network. Instead of purpose-built hardware appliances of the past, recent generations of the core have been primarily built as a set of disaggregated NFs, following the microservice paradigm. A number of crucial control plane NFs play distinct, but important roles, including the Access and Mobility Function (AMF), Network Repository Function (NRF), Service Management Function (SMF), and Authentication Server Function (AUSF). However, this disaggregated design requires networking to interconnect NFs, following a cloud-native, service-based architecture. A 3GPP-compliant 5GC typically adopts the service-based interface (SBI) with HTTP/REST APIs for seamless inter-service communication. As we look to the future, it is critical that we avoid some of the overheads of such an interface, including data copying, context switching, and notification overheads. This will reduce latency and CPU consumption, which are important to meet the needs of future applications and scalability.

7.2.2 Emerging 5G application needs

Typical advanced use cases include multi-player online gaming, augmented reality (AR), autonomous vehicles (AV), and other low-latency applications that require ubiquitous wireless connectivity. Furthermore, with the widespread use of cellular networks as replacements for traditional fixed wireline infrastructure, such as cable and fiber networks,

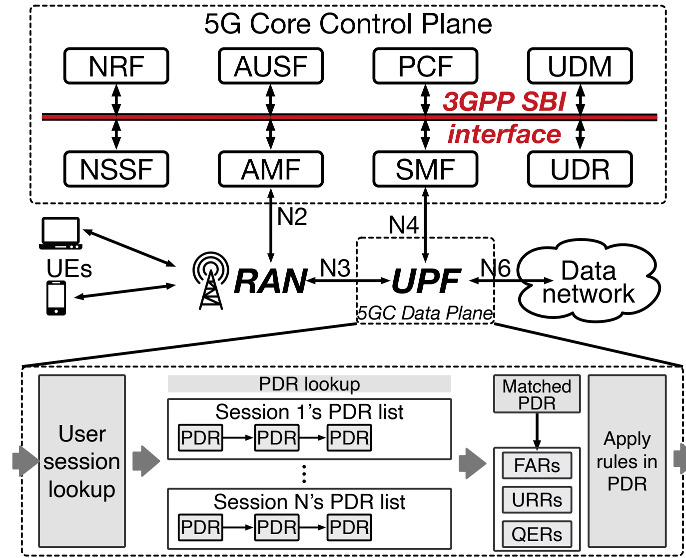


Figure 7.1: The architecture of 5GC control and data planes. The 5G UPF performs PDR lookup (based on linked list) to enable data packet forwarding between the RAN and data network.

we anticipate challenges arising from complex firewalls and home gateways associated with future cellular networks.

Emerging applications such as AR and AV can be very helpful [95], enhancing the perception of the real world with virtual holograms. These applications need high bandwidth and have stringent constraints on latency [114]. It is critical therefore that the cellular core also support the bandwidth needs of all the connected UEs without being a dataplane bottleneck. It is equally critical that the cellular network not introduce significant latency, both in the data and control planes [79].

Multi-player games often have many application flows that need differentiated treatment of throughput and latency-sensitive sub-flows. A game streamer may also be sending each of her control data, video, and voice streams separately. Messaging video,

audio, and text between users (usually mediated by a server) may also need to be treated differently and even routed differently through the network. All of these suggest the need for more fine-grained packet detection and handling of each distinct flow, using appropriate Differentiated Services Code Point markings, rather than treating all flows of a UE in the same way.

In the future, a single UE (or a cellular home gateway) may have multiple concurrently active IP flows. Thus, we would need a flow-specific firewall. Flows may have different security and QoS levels, requiring different PDRs in the UPF to support their processing. Having the UPF support feature-rich firewall rules to handle packet filtering for specific traffic, while desirable, can lead to increasing demand for PDRs for a single user session. A cellular home gateway will likely increase the number of PDRs for a single user session. A 5G home gateway that connects multiple mobile devices would behave as a “virtual” UE that registers many PDRs for the multiple concurrent flows of each user session.

These use cases will inevitably increase the number of PDRs registered in the UPF for an individual user session, increasing search complexity if it is based on a linked-list-based PDR-lookup.

7.2.3 Limitations of 3GPP Cellular Core

Slow 3GPP SBI: Industry direction has been the disaggregation of the cellular core network components. For example, AT&T’s mobile core network spans more than 60 cloud-native virtual NFs from 15 different vendors [120]. Several distinguishing characteristics of the cellular environment present challenges for such a disaggregation: (1) tight coupling

and timing constraints between the control and data planes; (2) frequent and increasing control plane activities that impact data plane handling, driven by emerging use cases such as IoT and frequent handovers, especially with small cells. Further, using the SBI for communication involves using JSON as the serialization format, which makes control actions (e.g., GTP tunnel setup, state synchronization) very expensive. To provide low-latency access, it is crucial to accelerate control plane procedures by minimizing communication overheads among different NFs.

Poor scalability - PDR Rule Lookup: As shown in Figure 7.1, following the 3GPP specification, the UPF organizes the registered PDRs in a linked list [130], in descending order of priority. For each user session in the UPF, a PDR list defines the packet processing criteria (*e.g.,* forwarding, buffering) of flows in user sessions. The UPF traverses the PDR list for that session to classify each packet. The matching PDR provides pointers to subsequent rules, *e.g.,* FARs (Forwarding Action Rules) and QERs (QoS Enforcement Rules), which together determine the action to be taken on the packet.

The current design may be adequate for common 5G use cases, where each user session typically only has a few PDRs. The complexity of the PDR search is minimal. However, as cellular networks further evolve (*e.g.,* 6G), more advanced use cases may significantly increase the number of PDRs to tens or even hundreds of entries for a single user session, leading to long PDR search delays, thus increasing data plane latency. It is important to have faster PDR lookup mechanisms.

7.3 Improved Control Plane Design in L^{2}5GC+

We begin with an overview of the L^{2}5GC+ and describe the key building blocks for developing a high-performance communication paradigm using shared-memory processing, while providing the necessary synchronous I/O primitives to replace the kernel-based 3GPP SBI. We then discuss in detail how to build an SBI on shared memory from the bottom-up. This includes asynchronous shared-memory processing over the DPDK, a POSIX-like synchronous I/O interface, and how we can use the POSIX-like APIs to build a shared-memory-based SBI. We then implement a seamless port of the 5GC control plane NFs from the baseline free5GC code-base to L^{2}5GC+.

We describe concurrent user session support in the L^{2}5GC+, including connection establishment, connection tear down, and user session management during data transfer between control plane NFs in L^{2}5GC+.

7.3.1 Overview of L^{2}5GC+'s SBI

Fig. 7.2 depicts the architecture of L^{2}5GC+'s control plane and SBI, which takes advantage of shared memory processing in userspace for data sharing between control plane NFs. This avoids expensive CPU data copy overheads, protocol processing, context switch, serialization, and deserialization, which are all incurred by the currently recommended 3GPP SBI. In the userspace of each worker node, L^{2}5GC+ dedicates a shared memory pool and adopts an NF manager to support shared memory processing. Information exchange is performed by message descriptor delivery between NFs. The per-node NF manager is responsible for managing shared memory (*e.g.,* initialization and removal) and interacts with

the protocol stack to provide a "one-time", consolidated protocol processing when inter-node communication is needed. Our current implementation utilizes the kernel protocol stack for inter-node communication as it is robust and proven. However, high-performance inter-node transport protocols, such as RDMA, may be desirable and is the subject of our future work.

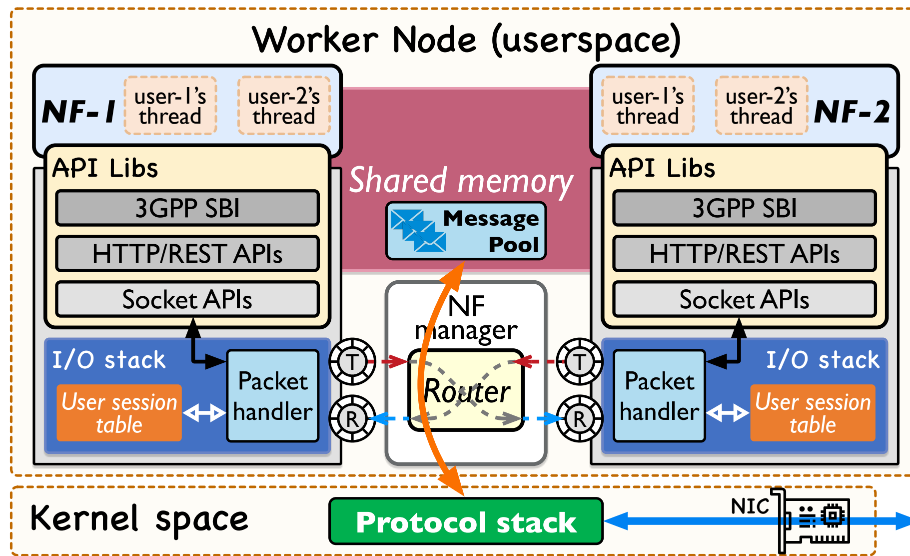


Figure 7.2: An architectural overview of L^{2}5GC+'s control plane and SBI. Each control plane NF can support multiple user sessions (each represented as a distinct thread) concurrently.

Each L^{2}5GC+ NF uses our newly developed I/O stack [184] for shared memory communication between other NFs that are located on the same node. The stack provides a shared memory I/O interface with a set of *asynchronous* communication primitives, utilizing DPDK [50]. A packet handler in the I/O stack deals with incoming/outgoing messages (essentially descriptor exchanges). L^{2}5GC+ also exploits the *lock-free communication* of X-IO [184] to avoid the need for any potential locks for multiple-producer, multiple-consumer communication. Such a communication pattern commonly exists in 5GC control plane.

Connection handling (*e.g.,* establishment/teardown) messages are processed by the packet handler for connection management tasks. These are important extensions in L^{2}5GC+ that help us to support scalable user sessions going beyond the capability of the previous L^{2}5GC implementation. A local connection table in the I/O stack maintains the connection state. The connection related to a data message is identified by looking up the connection table based on the IP 4-tuple (source IP and port number, destination IP and port number). The packet handler directs the message to the right connection endpoint (*i.e.,* the corresponding user thread) in this I/O stack.

The primitives exposed for *asynchronous* shared memory I/O by the I/O stack in L^{2}5GC [130] are not 3GPP-compliant. Therefore, it requires extensive refactoring of the baseline free5GC [58] implementation. Thus, we seek to overcome this deficiency. L^{2}5GC+ introduces an API library (API lib for short), to provide the necessary *synchronous* I/O primitives, enabling the interface to be compliant with 3GPP.

As shown in Fig. 7.2, the top layer NF code performs several control plane tasks across the SBI, using *synchronous* I/O. This then interacts with the *asynchronous* I/O stack below using the API lib, which includes a socket interface to directly interact with the underlying I/O stack for shared memory communication. A set of HTTP/REST APIs are provided on top of the socket interface. L^{2}5GC+ utilizes these APIs to create a 3GPP-compliant SBI. L^{2}5GC+'s SBI using shared memory has the same semantics as the 3GPP's SBI, just like free5GC [58], for easy portability to L^{2}5GC+.

7.3.2 Shared memory management in L^{2}5GC+

L^{2}5GC+ depends on the NF manager to manage the shared memory pool. During the initialization of the L^{2}5GC+ environment, an NF manager is created on a designated worker node. The NF manager then creates a certain number of buffers within the shared memory pool to be utilized as shareable backends for exchanging control plane messages between L^{2}5GC+ NFs.

We extensively use DPDK's libraries [50] to implement shared memory management in L^{2}5GC+. For lifecycle management (*i.e.,* creation/recycle/destroy) of the shared memory buffer, we utilize DPDK's Mempool Library [52]. To enforce access control of the shared memory pool and to prohibit unauthorized access, we leverage the security domain design that is widely adopted in DPDK-based shared memory frameworks [130, 181, 185, 225], depending on DPDK's Environment Abstraction Layer [56] and multi-process support [53] to provide the necessary memory isolation.

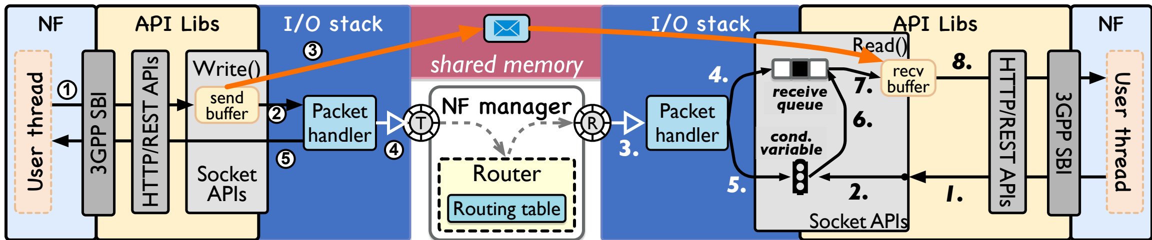


Figure 7.3: Synchronous I/O primitives from L^{2}5GC+'s socket APIs: Read() and Write(). Note that the circled number (*e.g.,* ①) represents the steps of Write() procedure (left half). Otherwise, it represents the steps of Read() procedure (right half).

7.3.3 Message descriptor delivery in $L^2$5GC+

A *lock-free* descriptor delivery mechanism is the key element to derive the value of shared memory communication in $L^2$5GC+. As shown in the overview figure (Fig. 7.2), each NF is assigned a pair of producer/consumer rings in its I/O stack. The producer/consumer rings of the NF are only shared with the NF manager, thus ensuring a strict single-producer, single-consumer communication pattern, avoiding the need for locks. On the other side, the NF manager forwards the descriptor (based on IP 4-tuples) between the I/O stacks of different NFs.

7.3.4 Building the SBI over shared memory: detailed design

We establish $L^2$5GC+'s 3GPP-compliant SBI starting from the *asynchronous* shared memory I/O adopted by $L^2$5GC, which is neither 3GPP-compliant nor scalable. We build an *asynchronous* shared memory I/O interface associated with a lock-free descriptor delivery mechanism into the I/O stack, thus offering these as raw I/O primitives to leverage shared memory processing.

We first introduce $L^2$5GC+'s API libs that add synchronous I/O support on top of the asynchronous I/O stack. The API libs adopt a layered design: (1) the bottom-layer of the library provides a set of POSIX-like socket APIs that directly interact with $L^2$5GC+'s I/O stack to leverage shared memory processing, while also providing basic synchronous I/O primitives; (2) the middle-layer library abstracts HTTP/REST APIs from the socket APIs, (3) these are then leveraged by the top-layer library to construct a 3GPP-compliant SBI.

This design choice was made primarily to facilitate ease of implementation and avoids re-implementing the entire stack — the implementation of upper-layer HTTP/REST APIs and 3GPP SBI can be ported from existing solutions (*e.g.,* free5GC) by simply replacing the lower-layer socket APIs, without being re-implemented from scratch.

Asynchronous shared memory I/O over DPDK: We construct the asynchronous shared memory I/O in `L`^{2}`5GC+` (also in L^{2}5GC) using DPDK's RTE RING [54] and Mempool APIs [52]. The basic I/O primitives that we use from DPDK to enable asynchronous shared memory processing include *rte_mempool_get()*, *rte_mempool_put()*, *rte_ring_enqueue()*, and *rte_ring_dequeue()*.

rte_mempool_get() and *rte_mempool_put()* are obtained from DPDK's Mempool lib. We use *rte_mempool_get()* to retrieve an empty memory buffer from the shared memory pool and return a descriptor (pointing to the retrieved buffer) to the caller NF. *rte_mempool_put()*, on the other hand, is used for recycling the buffer back to the shared memory pool.

rte_ring_enqueue() and *rte_ring_dequeue()* are obtained from DPDK's RTE Ring lib. *rte_ring_enqueue()* is specifically used for enqueuing the descriptor into the producer (TX) ring, while *rte_ring_dequeue()* is used for retrieving the descriptor from the consumer (RX) ring.

The asynchronous access mainly comes from the **non-blocking** nature of *rte_ring_enqueue()* and *rte_ring_dequeue()* APIs. The call to these APIs immediately returns, leading to a mismatch in the synchronous communication required by the upper-layer SBI.

Synchronous shared memory I/O: Following the design of X-IO [184], we abstract the synchronous I/O primitives of `L`^{2}`5GC+` into two socket APIs, Write() and Read(), maintain-

ing strict alignment with the POSIX-like socket APIs. We further add synchronous access by enforcing a blocking call to Write() and Read(), *i.e.,* the caller of the Write() and Read() API is blocked until the requested I/O task is accomplished.

The synchronous nature of the Write() API is achieved by blocking the caller thread until the message is moved from the send buffer (provided by caller thread) into the shared memory buffer. Fig. 7.3 (left half) shows how the Write() API interacts with the I/O stack to accomplish the transmission of a message: ① The caller thread initiates the Write() call with a send buffer input; ② The Write() call passes the send buffer to the packet handler in the I/O stack; ③ the packet handler then copies the message from the send buffer to the shared memory buffer. Note that, beforehand, the packet handler obtains an empty memory buffer and associated descriptor using *rte_mempool_get()* API; ④ The packet handler enqueues the descriptor to the producer (TX) ring; and then the packet handler ⑤ unblocks the Write() call to return control to the caller thread.

The synchronous Read() API is achieved by blocking the caller thread until the message is moved from the shared memory buffer to the receive buffer of the caller thread. We take advantage of the approach designed by X-IO [184] to enable the blocking Read() over the asynchronous shared memory I/O. There are two essential elements to block the caller of the Read(), including a condition variable [71] and a receive queue. Note that *each* user thread owns a dedicated condition variable as well as a receive queue for the sake of concurrent user session operations (§7.3.5).

The condition variable is utilized to suspend the caller thread of the Read() until its state is updated. Note that the condition variable is in effect only when its state is `TRUE`.

We use the receive queue to buffer descriptors whose message payload has not yet been transferred from the shared memory buffer to the receive buffer of the caller thread.

These two elements in the Read() interact with the asynchronous packet handler in the I/O stack to accomplish the blocking receive, as depicted in Fig. 7.3 (right half): (1.) When the caller thread initiates the Read() call, (2.) it is blocked on the condition variable (now set as `TRUE`). Subsequently, (3.) the I/O stack receives a descriptor from the NF on the other side, and (4.) enqueues the descriptor into the receive queue of the caller thread. (5.) The I/O stack updates the condition variable to `FALSE` to unblock the Read() call. (6.) The Read() call then dequeues the descriptor from the receive queue and it (7.) copies the message from the shared memory buffer to the receive buffer of caller thread. Finally, (8.) control returns to the caller thread. Note that Fig. 7.3 (right half) shows only the case when there is a single user session. Details of message reception for concurrent user sessions are given in §7.3.5.

Connection management: Similar to a POSIX-like socket interface (*e.g.,* socket APIs, HTTP/REST APIs, 3GPP SBI), the synchronous I/O interface in $\texttt{L}^2\texttt{5GC+}$ also requires pre-established connections to facilitate data transmission. We adopt the approach from X-IO [184] to manage the lifecycle of connections, including their establishment and teardown. Each NF's I/O stack maintains a local connection table, as shown in Fig. 7.2, which records essential connection-specific information, such as the IP 4-tuple of the source and destination NFs. Introducing the notion of a connection is a key extension of $\texttt{L}^2\texttt{5GC+}$ beyond its predecessor, $L^2$5GC, which allows $\texttt{L}^2\texttt{5GC+}$ to distinguish different user sessions, as described in §7.3.5.

Implementation of HTTP/REST APIs and 3GPP SBI in $\texttt{L}^2\texttt{5GC+}$: Our intention is to port the implementation of a 3GPP-compliant implementation like free5GC to $\texttt{L}^2\texttt{5GC+}$, to ensure that $\texttt{L}^2\texttt{5GC+}$ also conforms to the specifications, leveraging free5GC's development effort. As such, we choose Golang [70] to develop the API libs in $\texttt{L}^2\texttt{5GC+}$. Golang was the primary programming language that free5GC's control plane NFs are written in.

Since we take a layered design approach for our API libs, we only re-implement the POSIX-like socket interface in order to interact with our asynchronous shared memory I/O stack. We keep the upper layer HTTP/REST APIs and 3GPP SBI *unchanged*. This is achieved by replacing the imported Golang's "`net`" [63] package[3] with $\texttt{L}^2\texttt{5GC+}$'s socket API package. Since our socket APIs keep the same semantics as Golang, the porting is seamless. The cross-language support is described next.

Cross-language support: The I/O stack in each $\texttt{L}^2\texttt{5GC+}$ NF is developed in `C` language, ensuring optimal performance and reliability. Seamless interaction between the `C`-based I/O stack and the higher-layer API libs (in Golang) is achieved through Golang's built-in CGo interface [49]. The C-to-Go boundary crossing incurs minimal additional latency (approximately 70ns in our testbed), showcasing negligible impact on data exchange performance.

7.3.5 Concurrent user session support

Thread-based concurrency is commonly used in representative implementations of 5GC control plane NFs that support multiple simultaneous user sessions. Each user session is handled by a dedicated thread in the control plane NF instance (typically deployed as a Linux process within a virtualized sandbox, *e.g.,* container) to accomplish certain control

[3]Golang's "`net`" [63] package is the official package that offers various POSIX interface for network I/O.

plane procedures (*e.g.,* UE registration, handover). In order to differentiate user sessions during control plane messaging between NFs, we bind a specific connection for each user session. As shown in Fig. 7.4, (1) when the destination NF receives a message descriptor, (2) it looks up its local connection table in the I/O stack and finds the correct connection, *i.e.,* user session, to refer to. Subsequently, (3) the I/O stack can enqueue the descriptor to the connection's receive queue, thus ensuring that the message is correctly directed to the appropriate user session thread. This multiplexing/de-multiplexing allows seamless concurrent processing of messages in L^{2}5GC+'s control plane.

Concurrency control: L^{2}5GC+ adopts the implementation of 5GC from free5GC [58], which relies heavily on Golang as the primary programming language for the control plane NFs. The concurrency support in Golang is implemented by *goroutines* [41], which are essentially lightweight threads managed by the Go runtime. With the connection abstraction in L^{2}5GC+, we can support multiple user sessions using thread-based concurrency.

However, we observe that certain 5GC control plane events (*e.g.,* PDU session establishment) incur very frequent context switches. This is caused by the CPU 'thrashing' between multiple user sessions when they complete the same 5GC control plane events concurrently using a limited number of CPU cores on the node. Since each user session requires a dedicated thread for each of the 5GC NFs to accomplish the control plane event-related tasks, it results in frequent thread context switches. This adds additional delay to the event completion time. As L^{2}5GC+ and free5GC share the same control plane NF implementation, they both suffer from this performance loss.

To overcome the effect of ‘thrashing’, we introduce a concurrency control mechanism to limit the number of concurrent execution of certain events (*e.g.,* PDU session establishment) that are processed simultaneously by the 5GC control plane using the available CPU cores on the node. We implement a rate limiter at the AMF, which is the ingress point of the 5GC control plane. After the concurrency threshold is reached, additional PDU session establishment requests are queued at the AMF. We note that the threshold will likely depend on the number of available CPU cores and their capability, the complexity of the operations, and likely the mix of operations. We experimentally determined the suitable concurrency level of *16* in our current testbed.

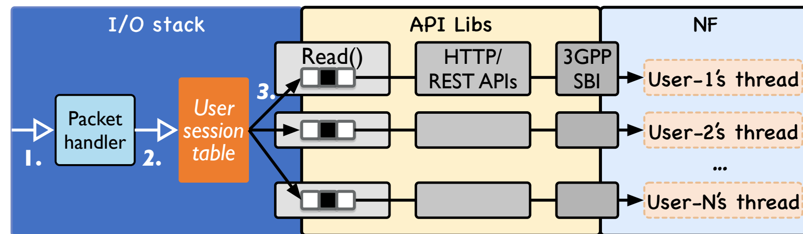


Figure 7.4: Concurrent user session support in $\texttt{L}^2\texttt{5GC+}$. Control plane messages to different user sessions are de-multiplexed at the I/O stack after user session table lookup.

7.3.6 Deployment strategy of $\texttt{L}^2\texttt{5GC+}$

$\texttt{L}^2\texttt{5GC+}$ adopts the same placement and scaling strategy as its predecessor, $\text{L}^2$5GC [130]. To harness the benefits of intra-node shared memory processing to minimize the latency of the 5GC control plane, it is important for $\texttt{L}^2\texttt{5GC+}$ NFs to be co-located on the same node.

In scenarios where inter-node communication becomes necessary (*e.g.,* when resource constraints prevent NF consolidation on a single node), $\texttt{L}^2\texttt{5GC+}$ relies on NF man-

agers to facilitate communication, by offering consolidated protocol processing. This can be achieved by using a kernel-based protocol stack or by employing a high-performance inter-node communication solution, such as RDMA with zero-copy data transfer primitives. This is part of our ongoing effort. This approach helps minimize the impact of kernel-based networking when inter-node communication is required.

It is important to note that the hybrid communication mode of $\texttt{L}^2\texttt{5GC+}$ (combining intra-node shared memory processing and inter-node kernel-based networking) ensures that the lower bound of performance remains no worse than a complete kernel-based networking solution. On the other hand, the typical as well as the upper bound performance of the hybrid communication mode of $\texttt{L}^2\texttt{5GC+}$ far outperforms kernel-based networking, especially when we consider node affinity to strategically place NFs, thus fully exploiting the benefits of shared memory processing. This hybrid approach offers a flexible and efficient solution that optimizes performance based on the specific deployment scenario and resource availability.

7.4 Fast and Scalable Packet Classification in $\texttt{L}^2\texttt{5GC+}$'s User Plane Function

7.4.1 Partitioning-based Versus Decision-tree-based Packet Classifiers

The PDR processing in the UPF is a packet classification problem. In order to achieve faster PDR lookup & updates for new cellular network use cases, we consider advanced packet classification mechanisms as alternatives to the linked list approach. We can extend existing packet classification mechanisms designed for routers to classify packets based on PDR Information Elements (IEs) without being limited to IP 5 tuples. We follow

the same packet processing pipeline suggested by 3GPP (Fig. 7.1) but replace the linked-list-based PDR lookup step with advanced packet classification mechanisms.

Typical packet classification mechanisms can be broadly classified into partitioning-based and decision-tree-based classifiers. Both achieve better performance than a linked-list-based approach since their search complexity is less than $O(n)$. Partitioning-based classifiers partition a set of PDRs into multiple sub-tables based on specific criteria, *e.g.,* a typical partitioning-based classifier, Tuple Space Search (TSS [201]), partitions PDRs with the same number of prefix bits in each IE field by combining them into the same sub-table. Therefore, the search complexity can be reduced. However, the classification time of partitioning-based classifiers may not be optimal because the number of partitioned sub-tables shows randomness. Given n PDRs, partitioned sub-tables ranges from 1 to n. In the worst case, n sub-tables need to be searched before finding a matched PDR, which results in the same search complexity as the linked list approach. In addition, partitioning-based classifiers (*e.g.,* TSS) typically organize each sub-table into a key-value hash table. Performing a lookup in a sub-table requires software hashing, which often takes more time than a lookup in a linked list. In the worst case, partitioning-based classifiers have a much higher classification latency than the linked list due to the overhead of software hashing.

Unlike partitioning-based classifiers, decision-tree-based classifiers show less variability in packet classification performance. The key for such decision-tree-based classifiers is to generate multidimensional classification trees, given a set of PDRs. Various greedy heuristics have been explored to generate efficient multidimensional classification trees (with logarithmic search time) to support high-speed packet classification [199, 186]. However,

decision-tree-based classifiers suffer from long rule update latency [222]. When constructing a multidimensional classification tree, the classifier cuts nodes starting from the 'root', which contains all the given PDRs. The classifier recursively splits the PDRs in each node until the number of PDRs within each leaf node is under a predefined threshold. Since different PDRs may intersect in some dimensions, splitting intersecting PDRs into different nodes is infeasible, resulting in a PDR being copied and added to multiple nodes. Multiple updates must be performed on each copy to ensure consistency among multiple copies of the same PDR, resulting in long delays for PDR updates. This vulnerability of decision-tree-based classifiers is avoidable in partitioning-based classifiers, as partitioning PDRs into multiple sub-tables does not lead to multiple copies of a PDR, thus achieving faster PDR updates.

7.4.2 PDR Lookup and Update in $\texttt{L}^2\texttt{5GC+}$

We examine different PDR management approaches (*e.g.,* Linear Search (PDR-LL), TSS (PDR-TSS) and ParititionSort (PDR-PS)) to reduce the searching complexity of large-scale PDR lookups in UPF-U, which we see as being critical for future-proofing the 5GC. Although PDR-TSS reduces the search complexity by partitioning rules into multiple rule sets (*i.e.,* sub-tables). It does not guarantee to reduce the number of resulting rule sets to be searched to find a match. This random behavior of PDR-TSS, combined with needing a software hash for searching a rule set, implies variable and potentially high overhead. PDR-PS avoids the weakness of PDR-TSS: **(1)** PDR-PS does not rely on hashing to search for a partitioned rule set (binary tree lookup). **(2)** PDR-PS, with PartitionSort's online ruleset partitioning, eliminates randomness and results in fewer partitioned rule sets,

yielding more consistent performance [222]. In addition, L^{2}5GC+'s UPF-U seeks to meet important requirements of operators, *e.g.,* avoiding DoS attacks [97]. PartitionSort helps to avoid TSS's vulnerability to DoS attack. We studied other state-of-the-art alternatives, *e.g.,* NeuroCuts [155], but lacking production data-sets for learning, we seek to use the option with the highest performance providing the needed flexibility. To accommodate a packetized 5GC, we employ a number of PDI IEs (up to 20) in the PDR to support rich functionality needed, including firewalls, NATs, and per-flow QoS treatment.

7.5 Evaluation

We evaluate the improved performance of L^{2}5GC+ using a real 5G testbed built using 3GPP-compliant commercial base stations, a variety of UEs (laptops with 5G dongles) and the L^{2}5GC+ running on commercial off-the-shelf (COTS) servers. We also consider a simulated 5G UE & RAN to evaluate the performance of L^{2}5GC+ as we scale up to more UEs. We compare L^{2}5GC+ with free5GC [58], an open-source, 3GPP-compliant 5GC implementation that has been used in many consortium-based 5G frameworks, *e.g.,* Magma [121], SD-CORE [44], and Aether [35].

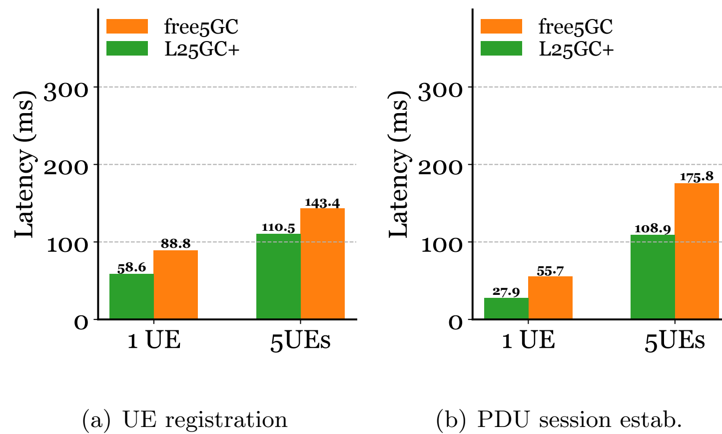
7.5.1 Control Plane Evaluation with *commercial* UEs and RAN

Commercial testbed setup: Fig. 7.5 shows the experimental setup of our real testbed, including the UE, RAN, and 5G core network (CN). Our 5G RAN contains an RU and a commercial CU/DU system built on an off-the-shelf computer system. We use a 'bare-metal' server to run the 5G core network, including both the control plane and data plane.

Figure 7.5: (Top) The logical topology, and (Bottom) a snapshot of commercial testbed setup in `L²5GC+`. RU: Radio Unit; DU: Distributed Unit; CU: Central Unit; CN: Core Network; PoE: Power over Ethernet; 5 laptops as UEs with 5G dongles

The CU/DU system connects to the 5GC control plane via the 3GPP N2 interface. The CU/DU system connects to the 5GC data plane (*i.e.,* UPF) via the 3GPP-specified N3 interface. The UPF of the 5GC connects to the data network via the N6 interface.

We choose the 5G small cell from Alpha Networks Inc. [67] as the RU. We use the commercially available CU/DU from AEWIN Technologies [66]. The server running the 5GC contains a 16-cores Intel Core i7 13700 CPU and 16G memory. We install two NICs on the 5GC server: a 10Gbps Intel X520-DA2 NIC used for N3 and N6 interface in the data plane, and a 2.5Gbps Realtek RTL8125BG NIC for the N2 interface in the control plane. We choose the 10Gbps NIC for the data plane for its higher bandwidth. We use a total of 5 laptops to emulate multiple UEs. Each laptop is equipped with an Apal 5G Dongle [51] to communicate with the RU in 5G RAN.

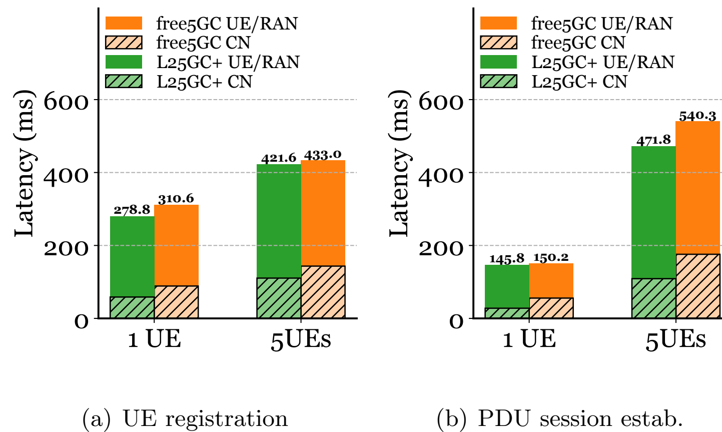


(a) UE registration (b) PDU session estab.

Figure 7.6: Latency from 5G CN, tested with ***commercial*** UEs and RAN.

Tested control plane events: We select a pair of representative control plane events to evaluate, including UE registration and the PDU (Packet Data Unit) session establishment that follows. During registration, the UE establishes its presence and identity on the 5GC

before accessing 5G services. A PDU session represents a logical connection between the UE and the 5GC for data communication. The PDU session establishment creates a dedicated data path between the UE and the 5GC data plane (*i.e.*, UPF) for handling data traffic. We evaluate L^{2}5GC+ with a single UE and also with 5 UEs running concurrently on the testbed to understand the ability of L^{2}5GC+ to support multiple user sessions in the 5G control plane (unlike L^{2}5GC which had limited support).



(a) UE registration (b) PDU session estab.

Figure 7.7: Total latency (including core network and UE/RAN) of control plane events tested with ***commercial*** UEs and RAN.

Fig. 7.6 shows the contribution to end-end latency by the 5G core network. The result demonstrates the performance benefits of L^{2}5GC+'s shared-memory-based SBI in a commercial testbed: When handling a single UE, L^{2}5GC+ has 1.51× lower latency than free5GC to complete a UE registration. L^{2}5GC+ also achieves 2× latency improvement compared to free5GC for PDU session establishment. When 5 UEs register simultaneously, L^{2}5GC+ saves 1.29× latency on average. L^{2}5GC+ lowers latency by 1.61× on average to complete 5 PDU session establishment events concurrently. These improvements come

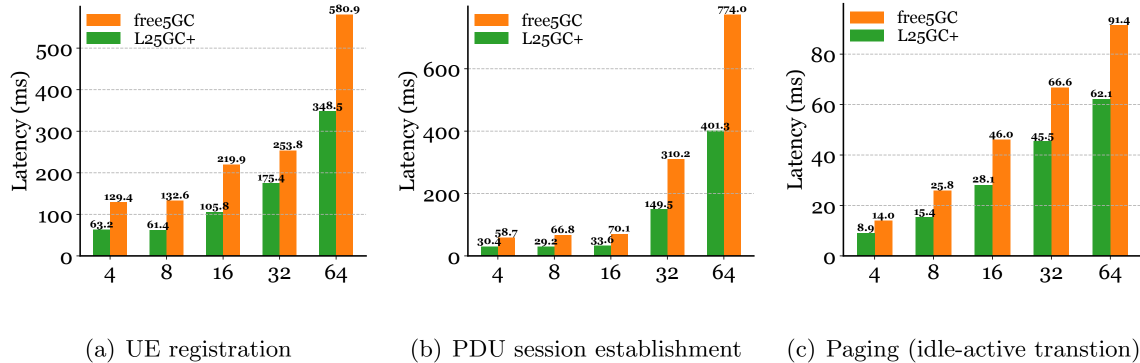
mainly from the use of shared memory processing in $L^2$5GC+, which incurs much lower communication overheads for control plane NFs to exchange messages, compared to the kernel-based SBI of free5GC.

In addition, we measure the "total" latency for different control plane events (in Fig. 7.7). This includes the latency contributed by the core network (named "CN") and the part contributed by the commercial UE/RAN. The somewhat slower UE and the disaggregated RAN system in our testbed reduces the relative impact of $L^2$5GC+'s improvements to this "total" latency. However, with higher-speed UEs (e.g., smart-phones) and improved RAN implementations, the significant reduction of the 5GC latency (the "CN" latency) due to our improvements will lower the overall cellular control plane latency.

7.5.2 Control Plane Evaluation with *simulated* UEs and RAN

We use the UE & RAN simulator from $L^2$5GC [130] to simulate user events, which allows us to scale up testing with more UEs. Throughout, we seek to understand the improved scalability of $L^2$5GC+ compared to the kernel-based SBI in free5GC, in handling multiple user sessions. We additionally evaluate the latency for paging events in the 5G control plane, where a UE transitions from idle to active state. A UE typically moves into an idle state to conserve (battery) energy, which is important for mobile devices and UEs such as IoT devices. When either a packet arrives at the 5GC or the UE has to transmit a (first) packet, the UE is "paged" to wake up the UE. The time it takes for the 5GC to complete this task and have the UE transition from idle to active has a direct impact on the latency experienced by that first packet. $L^2$5GC+ seeks to improve this latency in its control plane implementation. We vary the number of UEs from 4 to 64 and report the

total latency (as in Fig. 7.7) that includes the total latency contributed by both the core network and the simulated UE/RAN.



(a) UE registration (b) PDU session establishment (c) Paging (idle-active transtion)

Figure 7.8: Total latency of various control plane events with ***simulated*** UEs and RAN. *x-axis*: number of UEs.

Fig. 7.8 shows the total completion latency of different control plane events when multiple user sessions operate concurrently in the 5GC control plane. Compared to the kernel-based SBI in free5GC, L^{2}5GC+'s shared memory SBI shows a consistent reduction in latency as the number of UEs increases up to 64. L^{2}5GC+ reduces UE registration latency by 1.87× on average. L^{2}5GC+ also reduces the average PDU session establishment latency by 2.1× and average paging latency by 1.6×.

7.5.3 Data Plane Evaluation

We measure the performance improvements of L^{2}5GC+'s PDR classifier on our testbed, and compare it with free5GC. We show the improvement in data plane latency and throughput against the linked-list-based PDR classifier.

Experiment Setup: Our evaluation testbed consists of three Intel® Xeon® CPU E5-2697 v3 @ 2.60GHz servers running Ubuntu 20.04 with kernel 5.4.0-33-generic. Each server has an Intel 82599ES 10G Dual Port NIC. Server-1 and server-3 are configured as the RAN/UE and DN, respectively. Server-2 runs the `L`^{2}`5GC+` core, including all the NFs in Fig. 7.1. We use MoonGen [105] on server-1 and server-3 for generating uplink and downlink traffic.

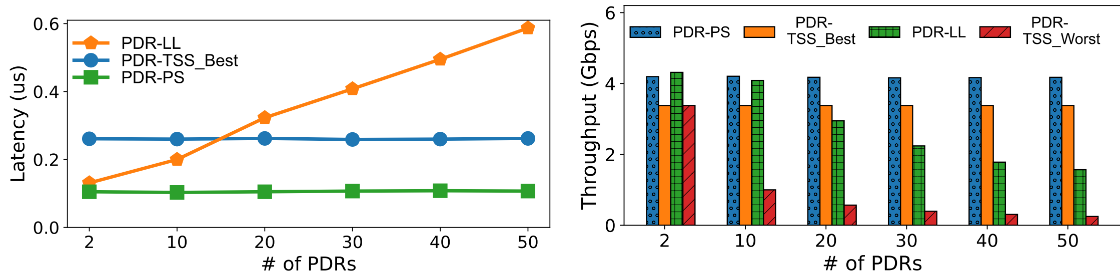


Figure 7.9: PDR lookup comparison: (a) PDR lookup latency with increasing # of PDR rules; (b) Throughput with increasing # of PDR rules (Packet size 68 bytes)

PDR lookup comparison

We compare the lookup performance between `PDR-TSS`, `PDR-PS`, and `PDR-LL`, with an increasing number of PDRs registered in the UPF-U. We extend ClassBench [208] to generate PDRs (including a total of 20 PDI IEs) for evaluation. Two different scenarios of `PDR-TSS` are studied: *Best case* (`PDR-TSS_Best`) and *Worst case* (`PDR-TSS_Worst`). For `PDR-TSS_Best`, given N PDR rules, only one sub-table is created, which indicates all the N PDRs are inserted to a single sub-table. One hash table lookup finds the target PDR. For `PDR-TSS_Worst`, the number of sub-tables equals the number of PDRs, which means each sub-table has only one PDR inserted. At most N sub-table lookups have to be performed to

find the matched PDR. In the PDR-TSS_Worst, we assume the match is in the last sub-table. For PDR-LL, we assume the packet randomly matches one of last 50% PDR entries in the list.

The comparison of PDR lookup latency and throughput is shown in Fig. 7.9(a) and 7.9(b). We omit the line "PDR-TSS_Worst" from Fig. 7.9(a) as it scales poorly, going up to $5.08us$ for 50 rules. This poor performance is primarily because of the heavyweight software hashing, and having to go through a number of PDR sub-tables. However, even with the increased number of PDRs installed in UPF-U, PDR-TSS_Best has a constant latency of $\sim 0.26us$. It also has much a lower latency than PDR-LL, which increases linearly with the number of rules. When there are 2 PDRs per session, PDR-LL may be acceptable as its throughput and latency are competitive. However, as we evolve to a packet-oriented environment, concerns about throughput degradation can be a major issue as the number of PDRs per session grows. PDR-PS achieves the best performance of all, for both latency and throughput. There is little degradation even with this growth in the number of PDRs per session. PDR-PS has the functionality of PDR-TSS, without its weaknesses. We have chosen to adopt PDR-PS classifier in $L^2$5GC+ for its superior performance.

PDR update comparison

Updating the PDR table is also a consideration, for session modification event, etc. We compare the PDR update performance of PDR-TSS, PDR-PS, and PDR-LL. We measure the average latency for a single PDR update with 50 repetitions. PDR-LL achieves an update latency of $0.38\mu s$, which is lower than PDR-TSS ($1.41\mu s$) and PDR-PS ($6.14\mu s$). Although PDR-TSS and PDR-PS have a higher update latency than PDR-LL, the difference

is not substantial. We choose `PDR-PS` for its reduced forwarding overhead and thereby the improved throughput, as seen in Fig. 7.9(b).

7.6 Conclusion

We presented $\texttt{L}^2\texttt{5GC+}$, a low-latency 5G control plane implementation. $\texttt{L}^2\texttt{5GC+}$ is an enhancement to our previous effort $L^2$5GC developed on top of OpenNetVM, a high-performance shared-memory NFV platform. $\texttt{L}^2\texttt{5GC+}$ makes several key extensions to $L^2$5GC, including support for concurrent user sessions using a 3GPP-compliant shared memory SBI. These two capabilities significantly improve the applicability of shared memory processing of the 5GC control plane, allowing 3GPP-compliant 5GC implementations such as free5GC to be seamlessly ported to $\texttt{L}^2\texttt{5GC+}$, while retaining the performance benefits of shared memory processing. Our evaluation using a testbed with commercial 5G UE and RAN components shows that $\texttt{L}^2\texttt{5GC+}$, with the help of its shared memory SBI, outperforms a kernel-based SBI implementation. $\texttt{L}^2\texttt{5GC+}$ significantly reduces the control plane latency for UE registration and PDU session establishment by $1.29\times$ and $1.61\times$, respectively, with 5 commercial UEs operating concurrently.

Chapter 8

Conclusions

High-performance cloud computing frameworks are critical for serverless computing and 5G systems to ensure quality of service to end users. However, serverless computing and 5G systems face different challenges when deploying high-performance cloud-based applications and need to be optimized for specific application requirements.

In our design of a high-performance, event-driven, and fair serverless computing framework, we first dissect the networking overheads that exists in current serverless frameworks. We studied the Container Network Interface (CNI), the defacto networking solution that has been widely used in various popular serverless platforms. Our analysis shows that kernel stack dominants the networking overheads between serverless function, while the in-kernel iptables accounts for more than 60% of the network packet processing cycles. Using eBPF can considerably speedup serverless networking compared to fully depending on heavyweight kernel protocol stack. This motivates us to design SPRIGHT, an event-driven serverless framework with a more streamlined network model and better scalability.

SPRIGHT demonstrated the effectiveness of event-driven capability for reducing resource usage and improving serverless function chaining in serverless cloud environments. With extensive use of eBPF-based event-driven capability in conjunction with high-performance shared memory processing, SPRIGHT achieves up to 5× throughput improvement, 53× latency reduction, and 27× CPU usage savings compared to Knative when serving a complex web workload. Compared to an environment using DPDK for providing shared memory and zero-copy delivery, SPRIGHT achieves competitive throughput and latency while consuming 11× fewer CPU resources. Additionally, for intermittent request arrivals typical of IoT applications, SPRIGHT still improves the average latency by 16% while reducing CPU cycles by 41%, when compared to Knative using 'pre-warmed' functions. This makes it feasible for SPRIGHT to support several 'warm' functions with minimum overhead (since CPU usage is load-proportional), sidestepping the 'cold-start' latency problem. Across several typical serverless workloads, SPRIGHT shows higher dataplane performance while avoiding the inefficiencies of current open-source serverless environments, thus getting us closer to meeting the promise of serverless computing. In addition, SPRIGHT saves 32% startup latency for a single function pod compared to Knative, which is an ideal capability for serverless computing. In addition, our proposed placement engine in Mu can ideally account for heterogeneity and fairness across competing serverless functions, ensuring overall resource efficiency, and minimizing resource fragmentation. Results show that it improves fairness by more than 2× over the default Kubernetes placement engine.

The rapidly growing field of Federated Learning heavily relies on the cloud for its computational resources. LIFL demonstrates that an enhanced control plane design for

serverless computing can enable FL model aggregation to benefit from the cloud's elasticity while efficiently utilizing its resources. Additionally, LIFL's improved data plane design, which incorporates eBPF-based shared memory processing from SPRIGHT, accelerates networking within a hierarchy of serverless aggregators. Evaluation shows that LIFL is nearly 3× faster than the baseline serverless FL setup (using Knative) in achieving 70% model accuracy when training the ResNet-18 model. Additionally, LIFL is almost 6× more efficient than the baseline serverless FL setup.

To address the excessive networking overheads in current 5GC control plane that uses kernel-based SBI, we propose L25GC+, a low-latency 5G control plane implementation. L25GC+ is developed on top of OpenNetVM, a high-performance shared-memory NFV platform. L25GC+ makes several key enhancements, including support for concurrent user sessions using a 3GPP-compliant shared memory SBI. These two capabilities significantly improve the applicability of shared memory processing of the 5GC control plane, allowing 3GPP-compliant 5GC implementations such as free5GC to be seamlessly ported to L25GC+, while retaining the performance benefits of shared memory processing. Our evaluation using a testbed with commercial 5G UE and RAN components shows that L25GC+, with the help of its shared memory SBI, outperforms a kernel-based SBI implementation. L25GC+ significantly reduces the control plane latency for UE registration and PDU session establishment by 1.29× and 1.61×, respectively, with 5 commercial UEs operating concurrently. We also explored different ways to improve the packet classification mechanisms in the UPF, minimizing PDR lookup and update latency. We compare the PartitionSort-based PDR classifier in the 5G UPF with two alternatives: a linked-list-based classifier and

a TSS-based classifier. The PartitionSort-based PDR classifier consistently achieves the lowest latency and highest throughput of all.

Bibliography

[1] Apache kafka. https://kafka.apache.org. [ONLINE].

[2] Building large clusters. https://kubernetes.io/docs/setup/best-practices/cluster-large/.

[3] Canal. https://github.com/projectcalico/canal.

[4] Cloud native computing foundation. https://www.cncf.io/.

[5] Cluster networking. https://kubernetes.io/docs/concepts/cluster-administration/networking/.

[6] Container network interface - networking for linux containers. https://github.com/containernetworking/cni.

[7] The container network model. https://github.com/moby/libnetwork/blob/master/docs/design.md#the-container-network-model.

[8] Contiv-vpp. https://github.com/contiv/vpp.

[9] iperf3 container. https://hub.docker.com/repository/docker/shixiongqi/iperf3.

[10] Knative serving component. https://knative.dev/docs/reference/serving-api/.

[11] Kubernetes: Lifecycle of a pod. https://dzone.com/articles/kubernetes-lifecycle-of-a-pod.

[12] nginx. https://nginx.org/en/.

[13] Pause container. https://hub.docker.com/r/google/pause/.

[14] Romana. https://github.com/romana/romana.

[15] Understanding pods. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/#understanding-pods.

[16] What is edge cloud? https://www.ciena.com/insights/what-is/What-is-Edge-Cloud.html.

[17] Choosing a CNI Network Provider for Kubernetes. https://chrislovecnm.com/kubernetes/cni/choosing-a-cni-provider/, 2017. [ONLINE].

[18] Kubernetes Control Plane. https://kubernetes.io/docs/concepts/overview/components/, 2020. [ONLINE].

[19] 503 service unavailable. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/503, 2021. [ONLINE].

[20] Apache Camel. https://camel.apache.org/camel-k/1.10.x/kamelets/kamelets-user.html, 2021. [ONLINE].

[21] wrk: a HTTP benchmarking tool. https://github.com/wg/wrk, 2021. [ONLINE].

[22] Apache OpenWhisk. https://openwhisk.apache.org/, 2022. [ONLINE].

[23] AWS Serverless API. https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/sam-resource-api.html, 2022. [ONLINE].

[24] Chaining OpenFaaS functions. https://ericstoekl.github.io/faas/developer/chaining_functions/, 2022. [ONLINE].

[25] IBM Cloud Functions. https://cloud.ibm.com/functions/, 2022. [ONLINE].

[26] Istio Architecture. https://istio.io/latest/docs/ops/deployment/architecture/, 2022. [ONLINE].

[27] Istio Traffic Management. https://istio.io/latest/docs/concepts/traffic-management/, 2022. [ONLINE].

[28] Locust. https://locust.io/, 2022. [ONLINE].

[29] of-watchdog. https://github.com/openfaas/of-watchdog, 2022. [ONLINE].

[30] OpenFaaS. https://www.openfaas.com/, 2022. [ONLINE].

[31] OpenFaaS API Gateway / Portal. https://docs.openfaas.com/architecture/gateway/, 2022. [ONLINE].

[32] OpenWhisk - Creating action sequences. https://github.com/apache/openwhisk/blob/master/docs/actions.md#creating-action-sequences, 2022. [ONLINE].

[33] OpenWhisk Composer. https://github.com/apache/openwhisk-composer, 2022. [ONLINE].

[34] SPRIGHT. https://github.com/ucr-serverless/spright.git, 2022. [ONLINE].

[35] Aether. https://opennetworking.org/aether/, 2023. [ONLINE].

[36] Autoscaling - Knative. https://knative.dev/docs/serving/autoscaling/, 2023. [ONLINE].

[37] Autoscaling - OpenFaaS. https://docs.openfaas.com/architecture/autoscaling/, 2023. [ONLINE].

[38] BPF maps. https://docs.kernel.org/bpf/maps.html, 2023. [ONLINE].

[39] extended Berkeley Packet Filter. https://ebpf.io/, 2023. [ONLINE].

[40] Fate: An Industrial Grade Federated Learning Framework. https://fate.fedai.org/, 2023. [ONLINE].

[41] How To Run Multiple Functions Concurrently in Go. https://www.digitalocean.com/community/tutorials/how-to-run-multiple-functions-concurrently-in-go, 2023. [ONLINE].

[42] Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/, 2023. [ONLINE].

[43] Open Federated Learning (OpenFL) - An Open-Source Framework For Federated Learning. https://github.com/intel/openfl, 2023. [ONLINE].

[44] SD-Core. https://opennetworking.org/sd-core/, 2023. [ONLINE].

[45] https://github.com/nycu-ucr/l25gc, 2024. [ONLINE].

[46] Apache Benchmark. https://httpd.apache.org/docs/2.4/programs/ab.html, 2024. [ONLINE].

[47] AWS Lambda. https://aws.amazon.com/lambda/, 2024. [ONLINE].

[48] BPF-HELPERS - list of eBPF helper functions. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html, 2024. [ONLINE].

[49] Cgo. https://pkg.go.dev/cmd/cgo, 2024. [ONLINE].

[50] Data Plane Development Kit. https://www.dpdk.org/, 2024. [ONLINE].

[51] Dongle - APAL. https://www.apaltec.com/dongle/, 2024. [ONLINE].

[52] DPDK Mempool Library. https://doc.dpdk.org/guides/prog_guide/mempool_lib.html, 2024. [ONLINE].

[53] DPDK Multi-process Support. https://doc.dpdk.org/guides/prog_guide/multi_proc_support.html, 2024. [ONLINE].

[54] DPDK Ring Library. https://doc.dpdk.org/guides/prog_guide/ring_lib.html, 2024. [ONLINE].

[55] eBPF XDP: The Basics and a Quick Tutorial. https://www.tigera.io/learn/guides/ebpf/ebpf-xdp/, 2024. [ONLINE].

[56] Environment Abstraction Layer. https://doc.dpdk.org/guides/prog_guide/env_abstraction_layer.html, 2024. [ONLINE].

[57] Flame: a federated learning system for the edge. https://github.com/cisco-open/flame, 2024. [ONLINE].

[58] free5GC. https://github.com/free5gc/free5gc, 2024. [ONLINE].

[59] Knative. https://knative.dev, 2024. [ONLINE].

[60] Knative Eventing. https://knative.dev/docs/eventing/, 2024. [ONLINE].

[61] Knative Serving. https://knative.dev/docs/serving/, 2024. [ONLINE].

[62] MQTT Version 5.0. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html, 2024. [ONLINE].

[63] net. https://pkg.go.dev/net, 2024. [ONLINE].

[64] NGINX. https://www.nginx.com/, 2024. [ONLINE].

[65] Online Boutique by Google. https://github.com/GoogleCloudPlatform/microservices-demo, 2024. [ONLINE].

[66] Performance Network System - SCB-1921B. https://www.alphanetworks.com/index.php/en/product_detail/a3caef706423d652, 2024. [ONLINE].

[67] Products: 5G Small Cell - Alpha Networks Inc. https://www.alphanetworks.com/index.php/en/product_detail/a3caef706423d652, 2024. [ONLINE].

[68] Project Calico. https://www.tigera.io/project-calico/, 2024. [ONLINE].

[69] Ring Library. https://doc.dpdk.org/guides/prog_guide/ring_lib.html, 2024. [ONLINE].

[70] The Go Programming Language. https://go.dev/, 2024. [ONLINE].

[71] Using condition variables. https://www.ibm.com/docs/en/aix/7.2?topic=programming-using-condition-variables, 2024. [ONLINE].

[72] Ahmed M. Abdelmoniem, Atal Narayan Sahu, Marco Canini, and Suhaib A. Fahmy. Refl: Resource-efficient federated learning. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 215–232, New York, NY, USA, 2023. Association for Computing Machinery.

[73] Marcelo Abranches, Oliver Michel, and Eric Keller. Getting back what was lost in the era of high-speed software packet processing. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 228–234, 2022.

[74] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[75] Mukhtiar Ahmad, Syed Usman Jafri, Azam Ikram, Wasiq Noor Ahmad Qasmi, Muhammad Ali Nawazish, Zartash Afzal Uzmi, and Zafar Ayyub Qazi. A low latency and consistent cellular control plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 648–661. ACM, 2020.

[76] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, 2018. USENIX Association.

[77] Sarah Allen, C Aniszczyk, C Arimura, et al. Cncf serverless whitepaper, 2018.

[78] Giuseppe Amato, Fabio Carrara, Fabrizio Falchi, Claudio Gennaro, and Claudio Vairo. Car parking occupancy detection using smart camera networks and deep learning. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 1212–1217, 2016.

[79] Kittipat Apicharttrisorn, Bharath Balasubramanian, Jiasi Chen, Rajarajan Sivaraj, Yi-Zhen Tsai, Rittwik Jana, Srikanth Krishnamurthy, Tuyen Tran, and Yu Zhou. Characterization of multi-user augmented reality over cellular networks. In *2020 17th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*, pages 1–9, 2020.

[80] Ryan Bankston and Jinhua Guo. Performance of container network technologies in cloud environments. In *2018 IEEE International Conference on Electro/Information Technology (EIT)*, pages 0277–0283. IEEE, 2018.

[81] Priscilla Benedetti, Mauro Femminella, Gianluca Reali, and Kris Steenhaut. Experimental analysis of the application of serverless computing to iot platforms. *Sensors*, 21(3), 2021.

[82] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[83] Matteo Bertrone, Sebastiano Miano, Jianwen Pi, Fulvio Risso, and Massimo Tumolo. Toward an ebpf-based clone of iptables. *Netdev'18*, 2018.

[84] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110, 2018.

[85] Dimitri P Bertsekas, Robert G Gallager, and Pierre Humblet. *Data networks*, volume 2. Prentice-Hall International New Jersey, 1992.

[86] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 153–167, New York, NY, USA, 2021. Association for Computing Machinery.

[87] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloé Kiddon, Jakub Konečný, Stefano Mazzocchi, Brendan McMahan, Timon Van Overveldt, David Petrou, Daniel Ramage, and Jason Roselander. Towards federated learning at scale: System design. In *Proceedings of Machine Learning and Systems*, volume 1, pages 374–388, 2019.

[88] Carsten Bormann, Angelo P Castellani, and Zach Shelby. Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62–67, 2012.

[89] Kevin Butler, Toni R Farley, Patrick McDaniel, and Jennifer Rexford. A survey of bgp security issues and solutions. *Proceedings of the IEEE*, 98(1):100–122, 2009.

[90] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[91] Lee Calcote. The container networking landscape: Cni from coreos and cnm from docker. `https://thenewstack.io/container-networking-landscape-cni-coreos-cnm-docker/`.

[92] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serverless programming (function as a service). In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2658–2659. IEEE, 2017.

[93] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The server is dead, long live the server: Rise of serverless computing, overview of current state and future trends in research and industry. *arXiv preprint arXiv:1906.02888*, 2019.

[94] Mohak Chadha, Anshul Jindal, and Michael Gerndt. Towards federated learning using faas fabric. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, page 49–54, New York, NY, USA, 2021. Association for Computing Machinery.

[95] Jiasi Chen, K.K. Ramakrishnan, Aditya Dhakal, and Xukan Ran. Networked architectures for localization-based multi-user augmented reality. *IEEE Communications Magazine*, pages 10–11, 2023.

[96] Cilium. https://cilium.io/.

[97] Levente Csikor, Dinil Mon Divakaran, Min Suk Kang, Attila Kőrösi, Balázs Sonkoly, Dávid Haja, Dimitrios P Pezaros, Stefan Schmid, and Gábor Rétvári. Tuple space explosion: A denial-of-service attack against a software packet classifier. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 292–304, 2019.

[98] Harshit Daga, Jaemin Shin, Dhruv Garg, Ada Gavrilovska, Myungjin Lee, and Ramana Rao Kompella. Flame: Simplifying topology extension in federated learning. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, 2023.

[99] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[100] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. Ecml: Improving efficiency of machine learning in edge clouds. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–6, 2020.

[101] Alexis Ducastel. Benchmark results of kubernetes network plugins (cni) over 10gbit/s network. https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-april-2019-4a9886efe9c4, 2019. [online].

[102] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[103] Justin Ellingwood. Comparing kubernetes cni providers: Flannel, calico, canal, and weave. https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/.

[104] Mohamed Elzohairy, Mohak Chadha, Anshul Jindal, Andreas Grafberger, Jianfeng Gu, Michael Gerndt, and Osama Abboud. Fedlesscan: Mitigating stragglers in serverless federated learning, 2022.

[105] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: a scriptable high-speed packet generator. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, pages 275–287. ACM, 2015.

[106] Md Hasanul Ferdaus, Manzur Murshed, Rodrigo N Calheiros, and Rajkumar Buyya. Virtual machine consolidation in cloud data centers using aco metaheuristic. In *European conference on parallel processing*, pages 306–317. Springer, 2014.

[107] Flannel. https://github.com/coreos/flannel/.

[108] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. Fast and efficient container startup at the edge via dependency scheduling. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.

[109] Alexander Fuerst and Prateek Sharma. Faascache: Keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 386–400, New York, NY, USA, 2021. Association for Computing Machinery.

[110] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: A serverless-first, light-weight wasm runtime for the edge. Middleware '20, page 265–279, New York, NY, USA, 2020. Association for Computing Machinery.

[111] Javad Ghaderi. Randomized algorithms for scheduling vms in the cloud. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[112] Javad Ghaderi, Yuan Zhong, and Rayadurgam Srikant. Asymptotic optimality of best-fit for stochastic bin packing. *ACM SIGMETRICS Performance Evaluation Review*, 42(2):64–66, 2014.

[113] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, pages 24–24, 2011.

[114] Moinak Ghoshal, Imran Khan, Z. Jonny Kong, Phuc Dinh, Jiayi Meng, Y. Charlie Hu, and Dimitrios Koutsonikolas. Performance of cellular networks on the wheels. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, page 678–695. ACM, 2023.

[115] Chaima Ghribi, Makhlouf Hadji, and Djamal Zeghlache. Energy efficient vm scheduling for cloud data centers: Exact allocation and migration algorithms. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 671–678. IEEE, 2013.

[116] Andreas Grafberger, Mohak Chadha, Anshul Jindal, Jianfeng Gu, and Michael Gerndt. Fedless: Secure and scalable federated learning using serverless computing. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 164–173, 2021.

[117] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[118] Yuanxiong Guo, Ying Sun, Rui Hu, and Yanmin Gong. Hybrid local sgd for federated learning with heterogeneous communications. In *International Conference on Learning Representations*, 2022.

[119] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2021. [ONLINE].

[120] Shawn Hakl. Improving the cloud for telcos: Updates of Microsoft's acquisition of AT&T's network cloud.

[121] Shaddi Hasan, Amar Padmanabhan, Bruce Davie, Jennifer Rexford, Ulas Kozat, Hunter Gatewood, Shruti Sanadhya, Nick Yurchenko, Tariq Al-Khasib, Oriol Batalla, Marie Bremner, Andrei Lee, Evgeniy Makeev, Scott Moeller, Alex Rodriguez, Pravin Shelar, Karthik Subraveti, Sudarshan Kandi, Alejandro Xoconostle, Praveen Kumar Ramakrishnan, Xiaochen Tian, and Anoop Tomar. Building Flexible, Low-Cost Wireless Access Networks With Magma. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1667–1681, Boston, MA, April 2023. USENIX Association.

[122] Haitham Hassanieh, Omid Abari, Michael Rodriguez, Mohammed Abdelghany, Dina Katabi, and Piotr Indyk. Fast Millimeter Wave Beam Alignment. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 432–445, New York, NY, USA, 2018. Association for Computing Machinery.

[123] Chaoyang He, Songze Li, Jinhyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, Xinghua Zhu, Jianzong Wang, Li Shen, Peilin Zhao, Yan Kang, Yang Liu, Ramesh Raskar, Qiang Yang, Murali Annavaram, and Salman Avestimehr. Fedml: A research library and benchmark for federated machine learning, 2020.

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

[125] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 54–66, 2018.

[126] Dzmitry Huba, John Nguyen, Kshitiz Malik, Ruiyu Zhu, Mike Rabbat, Ashkan Yousefpour, Carole-Jean Wu, Hongyuan Zhan, Pavel Ustinov, Harish Srinivas, Kaikai Wang, Anthony Shoumikhin, Jesik Min, and Mani Malek. Papaya: Practical, private, and scalable federated learning. In *Proceedings of Machine Learning and Systems*, volume 4, pages 814–832, 2022.

[127] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 445–458, Seattle, WA, 2014. USENIX Association.

[128] IBM. Creating serverless REST APIs. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-apigateway, 2022. [ONLINE].

[129] Intel. Multus. https://github.com/intel/multus-cni, 2020. [ONLINE].

[130] Vivek Jain, Hao-Tse Chu, Shixiong Qi, Chia-An Lee, Hung-Cheng Chang, Cheng-Ying Hsieh, K. K. Ramakrishnan, and Jyh-Cheng Chen. L25GC: A Low Latency 5G Core Network based on High-performance NFV Platforms. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 143–157, 2022.

[131] Vivek Jain, Sourav Panda, Shixiong Qi, and K. K. Ramakrishnan. Evolving to 6G: Improving the Cellular Core to lower control and data plane latency. In *2022 1st International Conference on 6G Networking (6GNet)*, pages 1–8. IEEE, 2022.

[132] Vivek Jain, Shixiong Qi, and KK Ramakrishnan. Fast function instantiation with alternate virtualization approaches. In *2021 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6. IEEE, 2021.

[133] K. R. Jayaram, Vinod Muthusamy, Gegi Thomas, Ashish Verma, and Mark Purcell. Adaptive aggregation for federated learning. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 180–185, 2022.

[134] K. R. Jayaram, Ashish Verma, Gegi Thomas, and Vinod Muthusamy. Just-in-time aggregation for federated learning. In *2022 30th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8, 2022.

[135] KR Jayaram, Vinod Muthusamy, Gegi Thomas, Ashish Verma, and Marc Purcell. Lambda fl: Serverless aggregation for federated learning. In *International Workshop on Trustable, Verifiable and Auditable Federated Learning*, page 9, 2022.

[136] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 152–166, New York, NY, USA, 2021. Association for Computing Machinery.

[137] Zhifeng Jiang, Wei Wang, Baochun Li, and Bo Li. Pisces: Efficient federated learning via guided asynchronous training. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 370–385, New York, NY, USA, 2022. Association for Computing Machinery.

[138] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. Ditto: Efficient serverless analytics with elastic parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 406–419, New York, NY, USA, 2023. Association for Computing Machinery.

[139] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. Hermod: Principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 289–305, New York, NY, USA, 2022. Association for Computing Machinery.

[140] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169, 2015.

[141] Narūnas Kapočius. Overview of kubernetes cni plugins performance. *Mokslas–Lietuvos ateitis/Science–Future of Lithuania*, 12, 2020.

[142] Narūnas Kapočius. Performance studies of kubernetes network solutions. In *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pages 1–6. IEEE, 2020.

[143] Tulja Vamshi Kiran Buyakar, Harsh Agarwal, Bheemarjuna Reddy Tamma, and Antony A. Franklin. Prototyping and Load Balancing the Service Based Architecture of 5G Core Using NFV. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 228–232, 2019.

[144] Kube-router. `https://www.kube-router.io/`.

[145] Fan Lai, Yinwei Dai, Sanjay Singapuram, Jiachen Liu, Xiangfeng Zhu, Harsha Madhyastha, and Mosharaf Chowdhury. Fedscale: Benchmarking model and system performance of federated learning at scale. In *International Conference on Machine Learning*, pages 11814–11827. PMLR, 2022.

[146] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*, pages 19–35. USENIX Association, 2021.

[147] Anusha Lalitha, Osman Cihan Kilinc, Tara Javidi, and Farinaz Koushanfar. Peer-to-peer federated learning on graphs, 2019.

[148] Jiaxin Lei, Manish Munikar, Kun Suo, Hui Lu, and Jia Rao. Parallelizing packet processing in container overlay networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 261–276, New York, NY, USA, 2021. Association for Computing Machinery.

[149] Joshua Levin and Theophilus A. Benson. Viperprobe: Rethinking microservice observability with ebpf. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–8, 2020.

[150] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, pages 2–es, 2007.

[151] Junfeng Li, Sameer G Kulkarni, KK Ramakrishnan, and Dan Li. Understanding open source serverless platforms: Design considerations and performance. In *Proceedings of the 5th International Workshop on Serverless Computing*, pages 37–42, 2019.

[152] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks. *Proceedings of Machine Learning and Systems*, 2:429–450, 2020.

[153] Tian Li, Maziar Sanjabi, Ahmad Beirami, and Virginia Smith. Fair resource allocation in federated learning. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[154] Yuanjie Li, Zengwen Yuan, and Chunyi Peng. A Control-plane Perspective on Reducing Data Access Latency in LTE Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 56–69, 2017.

[155] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *Proceedings of the ACM SIGCOMM 2019*, page 256–269. ACM, 2019.

[156] Ping-Min Lin and Alex Glikson. Mitigating cold starts in serverless platforms: A pool-based approach, 2019.

[157] Jiachen Liu, Fan Lai, Yinwei Dai, Aditya Akella, Harsha V. Madhyastha, and Mosharaf Chowdhury. Auxo: Efficient federated learning via scalable client clustering. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 125–141, New York, NY, USA, 2023. Association for Computing Machinery.

[158] Jiachen Liu, Fan Lai, Ding Ding, Yiwen Zhang, and Mosharaf Chowdhury. Venn: Resource management across federated learning jobs, 2023.

[159] Yu-Sheng Liu, Shixiong Qi, Po-Yi Lin, Han-Sing Tsai, K. K. Ramakrishnan, and Jyh-Cheng Chen. L25gc+: An improved, 3gpp-compliant 5g core for low-latency control plane operations. In *2023 IEEE 12th International Conference on Cloud Networking (CloudNet)*, 2023.

[160] Heiko Ludwig, Nathalie Baracaldo, Gegi Thomas, Yi Zhou, Ali Anwar, Shashank Rajamoni, Yuya Ong, Jayaram Radhakrishnan, Ashish Verma, Mathieu Sinn, et al. Ibm federated learning: an enterprise framework white paper v0. 1. *arXiv preprint arXiv:2007.10987*, 2020.

[161] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay ThakuR, Larry Peterson, Jennifer Rexford, and Oguz Sunay. A P4-Based 5G User Plane Function. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, SOSR '21, page 162–168, New York, NY, USA, 2021. Association for Computing Machinery.

[162] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282. PMLR, 20–22 Apr 2017.

[163] Jiayi Meng, Jingqi Huang, Y Charlie Hu, Yaron Koral, Xiaojun Lin, Muhammad Shahbaz, and Abhigyan Sharma. Characterizing and Modeling Control-Plane Traffic for Mobile Core Network. *arXiv preprint arXiv:2212.13248*, 2022.

[164] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. Securing linux with a faster and scalable iptables. *SIGCOMM Comput. Commun. Rev.*, 49(3):2–17, nov 2019.

[165] Microsoft. Azure - Function chaining in Durable Functions. `https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-sequence?tabs=csharp`, 2022. [ONLINE].

[166] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G. Kulkarni, Dan Li, Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 168–181, New York, NY, USA, 2021. Association for Computing Machinery.

[167] Jeffrey C Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[168] Ali Mohammadkhan, K. K. Ramakrishnan, and Vivek A. Jain. CleanG—Improving the Architecture and Protocols for Future Cellular Networks With NFV. *IEEE/ACM Transactions on Networking*, 28(6):2559–2572, 2020.

[169] John Nguyen, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Mike Rabbat, Mani Malek, and Dzmitry Huba. Federated learning with buffered asynchronous aggregation. In *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*, volume 151 of *Proceedings of Machine Learning Research*, pages 3581–3607. PMLR, 28–30 Mar 2022.

[170] Takayuki Nishio and Ryo Yonetani. Client selection for federated learning with heterogeneous resources in mobile edge. In *ICC 2019-2019 IEEE international conference on communications (ICC)*, pages 1–7. IEEE, 2019.

[171] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[172] Sourav Panda, K. K. Ramakrishnan, and Laxmi N. Bhuyan. Synergy: A SmartNIC Accelerated 5G Dataplane and Monitor for Mobility Prediction. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)*, pages 1–12, 2022.

[173] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, page 154–167, New York, NY, USA, 2021. Association for Computing Machinery.

[174] Youngki Park, Hyunsik Yang, and Younghan Kim. Performance analysis of cni (container networking interface) based container network. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 248–250. IEEE, 2018.

[175] Federico Parola, Fulvio Risso, and Sebastiano Miano. Providing telco-oriented network services with ebpf: the case for a 5g mobile gateway. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 221–225, 2021.

[176] Imtiaz Parvez, Ali Rahmati, Ismail Guvenc, Arif I. Sarwat, and Huaiyu Dai. A Survey on Low Latency Towards 5G: RAN, Core Network and Caching Solutions. *IEEE Communications Surveys & Tutorials*, 20(4):3098–3130, 2018.

[177] Christos-Alexandros Psomas and Jarett Schwartz. Beyond beyond dominant resource fairness: Indivisible resource allocation in clusters. *Tech Report Berkeley, Tech. Rep.*, 2013.

[178] Shixiong Qi, Sameer G. Kulkarni, and K. K. Ramakrishnan. Assessing container network interface plugins: Functionality, performance, and scalability. *IEEE Transactions on Network and Service Management*, 18(1):656–671, 2021.

[179] Shixiong Qi, Sameer G Kulkarni, and KK Ramakrishnan. Understanding container network interface plugins: design considerations and performance. In *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN*, pages 1–6. IEEE, 2020.

[180] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-Chin Wang, and K. K. Ramakrishnan. Spright: High-performance ebpf-based event-driven, shared-memory processing for serverless computing. *IEEE/ACM Transactions on Networking*, pages 1–16, 2024.

[181] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 780–794, 2022.

[182] Shixiong Qi, K. K. Ramakrishnan, and Jyh-Cheng Chen. L26gc: Evolving the low-latency core for future cellular networks. *IEEE Internet Computing*, 28(2):29–36, 2024.

[183] Shixiong Qi, K. K. Ramakrishnan, and Myungjin Lee. Lifl: A lightweight, event-driven serverless platform for federated learning. In *Proceedings of Machine Learning and Systems*, 2024.

[184] Shixiong Qi, Han-Sing Tsai, Yu-Sheng Liu, KK Ramakrishnan, and Jyh-Cheng Chen. X-IO: A High-performance Unified I/O Interface using Lock-free Shared Memory Processing. In *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, pages 107–115. IEEE, 2023.

[185] Shixiong Qi, Ziteng Zeng, Leslie Monis, and K. K. Ramakrishnan. MiddleNet: A Unified, High-Performance NFV and Middlebox Framework with eBPF and DPDK. *IEEE Transactions on Network and Service Management*, 2023.

[186] Yaxuan Qi, Lianghong Xu, Baohua Yang, Yibo Xue, and Jun Li. Packet classification algorithms: From theory to practice. In *IEEE INFOCOM 2009*, pages 648–656, 2009.

[187] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. Breakfast of champions: towards zero-copy serialization with nic scatter-gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 199–205, 2021.

[188] Red Hat, Inc. Understanding the eBPF networking features in RHEL. `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/assembly_understanding-the-ebpf-features-in-rhel-8_configuring-and-managing-networking`, 2024. [ONLINE].

[189] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and H. Brendan McMahan. Adaptive federated optimization, 2020.

[190] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. Fine-grained isolation for scalable, dynamic, multi-tenant edge clouds. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 927–942, 2020.

[191] Robert Ricci, Eric Eide, and CloudLab Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[192] Harmeet Sahni. The tale of two container networking standards: Cnm v. cni. `https://www.nuagenetworks.net/blog/container-networking-standards/`.

[193] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. Lukewarm serverless functions: Characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 757–770, New York, NY, USA, 2022. Association for Computing Machinery.

[194] Brandon Scheller. Best practices for resizing and automatic scaling in Amazon EMR. `https://aws.amazon.com/blogs/big-data/best-practices-for-resizing-and-automatic-scaling-in-amazon-emr/`, 2023. [ONLINE].

[195] Pavlos Sermpezis, Vasileios Kotronis, Petros Gigis, Xenofontas Dimitropoulos, Danilo Cicalese, Alistair King, and Alberto Dainotti. Artemis: Neutralizing bgp hijacking within a minute. *IEEE/ACM Transactions on Networking*, 26(6):2471–2486, 2018.

[196] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[197] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[198] Jaemin Shin, Yuanchun Li, Yunxin Liu, and Sung-Ju Lee. Fedbalancer: Data and pace control for efficient federated learning on heterogeneous clients. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 436–449, New York, NY, USA, 2022. Association for Computing Machinery.

[199] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, page 213–224, New York, NY, USA, 2003. Association for Computing Machinery.

[200] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 138–152, New York, NY, USA, 2021. Association for Computing Machinery.

[201] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, page 135–146, New York, NY, USA, 1999. Association for Computing Machinery.

[202] Kotikalapudi Sriram, Doug Montgomery, Brian Dickson, Keyur Patel, and Andrei Robachevsky. Methods for detection and mitigation of bgp route leaks. *draft-ietf-idr-route-leak-detection-mitigation-06*, 2017.

[203] Mark Stillwell, Frederic Vivien, and Henri Casanova. Virtual machine resource allocation for service hosting on heterogeneous distributed platforms. 2012.

[204] Alexander L Stolyar and Yuan Zhong. A large-scale service system with packing constraints: Minimizing the number of occupied servers. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):41–52, 2013.

[205] Alexander L Stolyar and Yuan Zhong. Asymptotic optimality of a greedy randomized algorithm in a large-scale service system with general packing constraints. *Queueing Systems*, 79(2):117–143, 2015.

[206] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 189–197. IEEE, 2018.

[207] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 311–327, New York, NY, USA, 2020. Association for Computing Machinery.

[208] David E Taylor and Jonathan S Turner. Classbench: A packet classification benchmark. *IEEE/ACM transactions on networking*, 15(3):499–511, 2007.

[209] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. Particle: Ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 16–29, New York, NY, USA, 2020. Association for Computing Machinery.

[210] Gurtzick Tobais. How kubernetes networking works – under the hood. `https://neuvector.com/network-security/advanced-kubernetes-networking/`. [ONLINE].

[211] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the open vswitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 245–257, 2021.

[212] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[213] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Deploying microservice based applications with kubernetes: experiments and lessons learned. In *2018 IEEE 11th international conference on cloud computing (CLOUD)*, pages 970–973. IEEE, 2018.

[214] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021.

[215] Ian-Chin Wang, Shixiong Qi, Elizabeth Liri, and K. K. Ramakrishnan. Towards a proactive lightweight serverless edge cloud for internet-of-things applications. In *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–4, 2021.

[216] Song Wang, Jingqi Huang, Xinyu Zhang, Hyoil Kim, and Sujit Dey. X-array: Approximating omnidirectional millimeter-wave coverage using an array of phased arrays. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, 2020.

[217] Weave. `https://github.com/weaveworks/weave`.

[218] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 206–212, 2021.

[219] Christopher R Wren, Yuri A Ivanov, Darren Leigh, and Jonathan Westhues. The merl motion detector dataset. In *Proceedings of the 2007 workshop on Massive datasets*, pages 10–14, 2007.

[220] Hiroki Yanagisawa, Takayuki Osogami, and Rudy Raymond. Dependable virtual machine allocation. In *2013 Proceedings IEEE INFOCOM*, pages 629–637. IEEE, 2013.

[221] Jiancheng Yang, Rui Shi, and Bingbing Ni. Medmnist classification decathlon: A lightweight automl benchmark for medical image analysis. In *IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pages 191–195, 2021.

[222] Sorrachai Yingchareonthawornchai, James Daly, Alex X. Liu, and Eric Torng. A sorted-partitioning approach to fast and scalable dynamic packet classification. *IEEE/ACM Transactions on Networking*, 26(4):1907–1920, 2018.

[223] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1489–1504, Boston, MA, April 2023. USENIX Association.

[224] Hao Zeng, Baosheng Wang, Wenping Deng, and Weiqi Zhang. Measurement and evaluation for docker container networking. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 105–108. IEEE, 2017.

[225] Ziteng Zeng, Leslie Monis, Shixiong Qi, and K. K. Ramakrishnan. Middlenet: A high-performance, lightweight, unified nfv and middlebox framework. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pages 180–188, 2022.

[226] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, pages 26–31, New York, NY, USA, 2016. ACM.