EPFL

Thèse n° 7108

Operating System and Network Co-Design for Latency-Critical Datacenter Applications

Présentée le 28 août 2020

à la Faculté informatique et communications
Laboratoire des Systèmes de Centres de Calcul
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Evangelos Marios KOGIAS

Acceptée sur proposition du jury

Prof. J. R. Larus, président du jury
Prof. E. Bugnion, directeur de thèse
Prof. C. Kozyrakis, rapporteur
Dr R. Bianchini, rapporteur
Prof. K. Argyraki, rapporteuse

École
polytechnique
fédérale
de Lausanne

2020

Νὰ εὔχεσαι νά ᾽ναι μακρὺς ὁ δρόμος.
Πολλὰ τὰ καλοκαιρινὰ πρωϊὰ νὰ εἶναι
ποὺ μὲ τί εὐχαρίστηση, μὲ τί χαρὰ
θὰ μπαίνεις σὲ λιμένας πρωτοειδωμένους·

— Κ. Π. Καβάφης

Hope the voyage is a long one.
May there be many a summer morning when,
with what pleasure, what joy,
you come into harbors seen for the first time;

— C. P. Cavafy

To my family…

Acknowledgements

This thesis is the final stop of a journey started five years ago, a journey that formed as a researcher and as a person in general. Throughout this journey I had several people that stood by me and supported me, including advisors, collaborators, friends, and family. This journey would not be possible without them and I wholeheartedly want to say a big thanks to them.
First and foremost I would like to thank my advisor Ed Bugnion. His huge systems insight deeply affected the way I approach and think about problems. Ed trusted me since the early days of my PhD and allowed me to take academic initiatives. Despite the big pressure his trust put on me, this trust helped me grow, form my identity as a researcher, make choices and stand by them, and advise many students. I will never forget our heated whiteboard sessions and my midnight Slack outbursts that he always dealt with profound patience.
Then, I would like to thank my thesis committee: James Larus, Katerina Argyraki, Christos Kozyrakis, and Ricardo Bianchini. They gave me a lot of constructive feedback that made this thesis much better but also influenced me throughout my PhD. Jim made me understand that simple questions are the hardest ones. Katerina served as my academic "aunt" during my time at EPFL and she was my go-to person during the downs of this journey. I had the luck to work with Christos during my first year at EPFL and he also deeply affected the way I approach research by making it clear that *"it's about the projects you finish, not the projects you start"*. Ricardo broadened my cloud horizons serving as my mentor during my MSR internship and taught me how to be a team player in a big project.
DCSL was never a big lab but I never felt alone in this journey. I am grateful to my labmates and our friends from VLSC for all the discussions we had throughout my PhD that helped this thesis come to its final form. Thank you very much Sam, Jonas, Mia, Dmitrii, Stuart, Sahand, David, Mayy. I want to specially thank my officemates George and Adrien. George was the senior student at DCSL when I joined and I learnt a lot from him. Adrien has always been there to discuss anything with his great ability to be direct, caring, and judgemental at the same time, while knowing which attribute of his to use in which discussion. I want to thank DCSL's mom, Maggy, for being extremely helpful with any issue I might have and dealing with our steam always with a big smile.
Having great mentors aspired me to try and become a good mentor myself. For that I would like to thank all my students both as a POCS TA and the ones that did projects with me. Kontantine, Martin, Antoine, Ogier, Lisa, Paul, Guillaume, Mikael, Kosta, Adi, Matthieu, and Charline thanks for working with me. I also want to thank my students in the greek dance class and the greek school. Teaching how to dance to such a versatile group of people taught

me how to explain things differently and adjust to my audience.
Doing a PhD is an extremely stressful period in a student's life and having great friends is the only way out. I want to thank Lefteris and George for being close to me in every step from day zero till the end. I feel grateful to all my Lausanne friends and especially to Irene, Patricia, Helena, Hermina, Lana, Kiril, Dusan, Mark, Rishabh, Arseniy, Arash, Bogdan, and the greek gang Stella, Thanos, Panagiotis, Panos, Aggelos, Pavlos, Natali, Vangelis, Prodromos, Tolis, Melivia, Georgia. Thanks for all the discussions (academic and non-), the BBQs, the beers at Great Escape, the ski trips. I want to specifically thank my friend Matt for luring me intro the world of motorbikes. I, also, want to thank my friends in Greece who, despite being far, always make me feel as I never left. Special thanks to my NTUA and SourceLair friends Antonis M, Antonis K, Paris, Thanasis, and Christos, my friend Anastasis for all the songs we've played together, and my great childhood friends George and Yiannis.
I wouldn't be the person I am today without the unconditional love and support of my family. I feel deeply grateful to my parents, Thanasis and Dimitra, for the way they raised me, my dad for showing me that every obstacle can be overtaken if you stay calm, and my mom for letting me dream without borders, my sister Alexia for reminding me the importance of people skills that can be easily lost throughout the PhD and my sister Evita for empowering me with her strong and non-conventional personality.
Last but not least, I want to thank Vivi. Vivi came in my life when I was 14 and never left ever since. She has been next to me in all the ups and downs holding different roles, but being my best friend above all, while she never stopped showing me that home is where they love you. Vivi has left indelible marks on who I am and I hope she will continue doing so in the years to come. Thank you very much for riding along.

⋆ ⋆ ⋆ ⋆ ⋆

I would also like the thank the people and the organisations that financially supported this thesis and allowed me to work on problems I found challenging. My thesis was supported by a VMware research grant and the Microsoft Swiss Joint Research Center. I am also a grateful recipient of the IBM PhD fellowship.

Lausanne, August 2020 M. K.

Abstract

Datacenters are the heart of our digital lives. Online applications, such as social-networking and e-commerce, run inside datacenters under strict *Service Level Objectives* for their tail latency. Tight latency SLOs are necessary for such services to remain interactive and keep users engaged. At the same time, datacenters operate under a single administrative domain which enables the deployment of customized network and hardware solutions based on specific application requirements. Customization enables the design of datacenter-tailored and SLO-aware mechanisms that are more efficient and have better performance.

In this thesis we focus on three main datacenter challenges. First, latency-critical, in-memory datacenter applications have µs-scale services times and run on top of hardware infrastructure which is also capable of µs-scale inter-node round-trip times. Existing operating systems, though, were designed under completely different assumptions and are not ready for µs-scale computing. Second, the base of datacenter communications is *Remote Procedure Calls* (RPCs) that depend on a message-oriented paradigm, while TCP still remains widely-used for intra datacenter communications. The mismatch between TCP's bytestream-oriented abstraction and RPCs causes several inefficiencies and deteriorates tail latency. Finally, datacenter applications follow a scale-out paradigm based on large fan-out communication schemes. In such a scenario, tail latency becomes a critical metric due to the tail at scale problem. The two main factors that affect tail latency is interference and scheduling/load balancing decisions.

To deal with the above challenges we advocate for a co-design of network and operating system mechanisms targeting µs-scale tail optimisations for latency-critical datacenter applications. Our approach investigates the potential of pushing functionality to the network leveraging emerging in-network programmability features. Whenever existing abstractions fail to meet the µs-scale requirements or restrict our design space, we propose new ones given the design and deployment freedom the datacenter offers.

This thesis contributions can be split in three main parts. We, first, design and build tools and methodologies for µs-scale latency measurements and system evaluation. Our approach depends on a robust theoretical background in statistics and queueing theory. We, then, revisit existing operating system and networking mechanisms for TCP-based datacenter applications. We design an OS scheduler for µs-scale tasks, while we modify TCP to improve L4 load balancing, and provide an SLO-aware flow control mechanism. Finally, after identifying the problems related to TCP-based RPC services, we introduce a new transport protocol for datacenter RPCs and in-network policy enforcement that enables us to push functionality to

the network. We showcase how the new protocol improves the performance and simplifies the implementation of in-network RPC load balancing, SLO-aware RPC flow control, and application-agnostic fault-tolerant RPCs.

Keywords: datacenters, datacenter networking, remote procedure call, in-network compute, µs-scale computing, scheduling, load balancing, P4, state machine replication

Résumé

Les centres de données sont au coeur de notre vie digitale. Les services en ligne tels que les réseaux sociaux ainsi que le commerce en ligne tournent dans des centres de données sous des objectifs de niveau de services (SLO) très strictes sur leur latence de traîne (tail latency). Des SLOs étroites sur la latence de traîne sont nécessaires pour que de tels services restent interactifs et gardent leurs utilisateurs engagés. En même temps, les centres de données opèrent sous un seul domaine administratif, ce qui permet de déployer des solutions personnalisées au niveau du réseau ou du matérel basé sur les besoins spécifiques de certaines applications. La personalisation permet la conception de mechanismes qui sont adaptées au centre de données et qui tiennent compte du SLO, permettant ainsi de meilleurs performance et plus d'efficacité.

Dans cette thèse nous nous concentrons sur trois principaux défisdans les centre de données. Premièrement, les applications tenant en mémoire et ayant une latence critique ont des SLO au niveau du us et tournent sur de l'infrastructure matériel qui est aussi capable d'effectuer des allers-retours inter-noeuds au niveau du us. Cependant, les systèmes d'exploitation ont été conçus sous des hypothèses très différentes et ne sont pas prêts pour des calculs au niveau du us. Deuxièment, la base de toute communication dans un centre de données est le Remote Procedure Call (RPC) suivant un paradigme de communication orienté sur les messages, tandis que TCP reste largement utilisé pour la communication intra-centre de données. Le décalage entre l'abstraction d'un flot d'octets fourni par TCP et les RPCs causent plusieurs inefficacités et détériorent la latence de traîne. Finalement, les applications dans les centres de données suivent une architecture de mise-à-l'échelle horizontale basée sur des modèles de communication à large fan-out. Dans un tel scénario la latence de traîne devient une mesure critique à cause du problème de traîne à l'échelle. Deux des facteurs principaux qui affectent la latence de traîne sont l'interférence ainsi que les décisions d'ordonnancement et de répartition de charge.

Afin de résoudre aux défis précédents, nous proposons de co-concevoir des mechanisms au niveau des réseaux et des systèmes d'exploitations visant les optimisations au niveau du us pour la latence de traîne pour des applications ayant une latence de traîne critique. Notre approche étudie le potentiel de pousser des fonctionnalités dans le réseau en tirant parti des capacités émergeants de programmabilité du réseau. Quand les abstractions existantes n'arrivent pas à atteindre nos exigences au niveau du us ou restreignent notre espace de conception, alors nous en proposons des nouvelles à cause de la liberté de conception et de déploiement offertes dans un centre de données.

Résumé

Les contributions de cette thèse peuvent être répartis en 3 parties principales. Premièrement, nous concevons et construisons des outils et des méthodologies pour faire des mesures de latence et d'évaluation de systèmes au niveau du us. Notre approche dépend sur une base robuste de statistiques et de la théorie des files d'attente. Nous revisitons ensuite des mécanismes existantes au niveau des systèmes d'éxploitation et du réseau pour des applications basées sur TCP. Nous construisons un ordonnanceur pour système d'exploitation visant des tâches au niveau du us, ainsi qu'en modifiant TCP afin d'améliorer la répartition de charge L4 et fournissons un mechanism de contrôle de flux tenant compte du SLO. Finalement, après avoir identifié les problèmes liés au services RPC basés sur TCP, nous introduisons un nouveau protocole de transport pour des RPC dans les centres de données ainsi qu'un mechanism de mise en vigueur au niveau du réseau qui nous permet de pousser des fonctionalités dans le réseau. Nous démontrons comment le nouveau protocole améliore la performance et simplifie l'implémentation de répartition de charge des RPC dans le réseau, le contrôle de flux en tenant compte du SLO, ainsi que des RPC ayant des tolérances aux pannes et étant agnostique à une application spécifique.

Mots clés : centre de données, réseaux des centres de données, RPC, calcule dans le réseau, calcul au niveau du us, ordonnancement, répartition de charge, P4, réplication de machine d'état.

Contents

Acknowledgements **i**

Abstract (English/Français) **iii**

List of Figures **xiii**

List of Tables **xix**

1 Introduction **1**
1.1 RPCs in the Epicenter 2
1.2 The Need for Tail-Tolerant Systems 2
1.3 The Advent of In-Network Compute 3
1.4 Thesis Statement 3
1.5 Thesis Contributions 4
1.6 Thesis Organisation 8
1.6.1 Bibliographic Notes 9

2 Background and Challenges **11**
2.1 Datacenter Infrastructure and Applications 11
2.2 Datacenter Remote Procedure Calls 15
2.3 Flow Control 17
2.4 Replication 18
2.4.1 Replication for Fault-Tolerance 18
2.4.2 Replication for Scalability 19
2.5 Internode Load Balancing 20

I Understanding and Measuring Latency Accurately **21**

3 Thesis Methodology **23**
3.1 System Modelling 23
3.2 Synthetic Microbenchmarks with Known Boundaries 25

4 Lancet: A self-correcting Latency Measuring Tool **27**
4.1 Introduction 27

4.2 Background . . . 29
4.2.1 Taxonomy of tools . . . 30
4.2.2 Configuration burden . . . 32
4.2.3 Statistics Background . . . 33
4.3 Experiment Decomposition . . . 34
4.4 Design . . . 35
4.4.1 LANCET infrastructure . . . 35
4.4.2 Measurement options . . . 36
4.4.3 LANCET's self-correcting methodology . . . 37
4.5 Implementation . . . 39
4.6 Evaluation . . . 42
4.6.1 Experimental setup . . . 43
4.6.2 Benefits of hardware timestamping . . . 43
4.6.3 LANCET self-controlling dynamics . . . 44
4.6.4 Inter-Arrival distribution Impact . . . 45
4.6.5 Server characterization . . . 46
4.7 Related Work . . . 46
4.8 Capter Summary . . . 47

5 Switch LOad Generation (SLOG) 49
5.1 Introduction . . . 49
5.2 Design . . . 50

II Latency-Critical Systems Using Existing Abstractions 55

6 Zygos: Achieving Low Tail Latency for Microsecond-scale Networked Tasks 57
6.1 Introduction . . . 57
6.2 Queueing Analysis . . . 59
6.3 Experimental Methodology . . . 62
6.3.1 Approach and metrics . . . 63
6.3.2 Experimental Environment . . . 63
6.3.3 Evaluated Systems . . . 64
6.3.4 Baseline results . . . 64
6.4 Design . . . 65
6.4.1 Requirements . . . 65
6.4.2 ZYGOS High-level Design . . . 66
6.4.3 Ordering semantics in multi-threaded applications . . . 68
6.4.4 Eliminating Head-of-Line Blocking . . . 68
6.4.5 Inter-processor Interrupts . . . 69
6.5 Implementation . . . 70
6.6 Evaluation . . . 71
6.6.1 Synthetic micro-benchmarks . . . 71

6.6.2 Overheads of ZYGOS on tiny tasks: memcached . . . 74
6.6.3 A real application: Silo running TPC-C . . . 75
6.7 Discussion . . . 77
6.7.1 The impact of SLO on systems . . . 77
6.7.2 Using Application Hints . . . 77
6.8 Related Work . . . 79
6.9 Chapter Summary . . . 81

7 CRAB: Bypassing the Load Balancer Without Regrets 83
7.1 Introduction . . . 83
7.2 Motivation and Background . . . 85
7.2.1 Load Balancing Flavors . . . 86
7.2.2 Load Balancing Policies . . . 89
7.3 Design . . . 91
7.4 Implementation . . . 93
7.4.1 Load Balancing Middlebox . . . 94
7.4.2 Connection Redirection . . . 94
7.5 Evaluation . . . 95
7.5.1 CRAB Load Balancer Implementations . . . 96
7.5.2 Latency Overhead . . . 97
7.5.3 Throughput Scaling . . . 98
7.5.4 Load Balancing Policies . . . 98
7.6 Discussion . . . 100
7.7 Related work . . . 102
7.8 Chapter Summary . . . 103

8 SLO-Aware TCP Flow Control 105
8.1 Introduction . . . 105
8.2 Motivation and Background . . . 106
8.2.1 Existing TCP Flow Control . . . 107
8.2.2 TCP Flow Control in KBNets . . . 107
8.3 Design . . . 108
8.4 Implementation . . . 111
8.5 Evaluation . . . 113
8.6 Related Work . . . 115
8.7 Chapter Summary . . . 116

III New Abstractions for RPCs 117

9 Request Response Pair Protocol (R2P2) 119
9.1 Introduction . . . 119
9.2 R2P2: A transport protocol for RPCs . . . 120

9.2.1 Transport considerations 122
9.2.2 API 123
9.3 Implementation 124
9.4 Related work 124
9.5 Chapter Summary 125

10 In-network RPC Load Balancing with JBSQ 127
10.1 Introduction 127
10.2 Queueing Analysis 128
10.3 The Join-Bounded-Shortest-Queue Policy 129
10.4 JBSQ router design considerations 130
10.5 Implementation 131
10.5.1 Router - software implementation 132
10.5.2 P4/Tofino implementation 132
10.6 Evaluation 133
10.6.1 Router characterization 134
10.6.2 Synthetic Time Microbenchmarks 136
10.6.3 Multi-packet Requests Microbenchmark 137
10.6.4 Using R2P2 for server work conservation 137
10.6.5 Lucene++ 138
10.6.6 Redis 140
10.7 Related work 140
10.8 Chapter Summary 141

11 In-network SLO-Aware RPC Flow Control 143
11.1 Introduction 143
11.2 Motivation 144
11.3 Design 145
11.3.1 SVEN Dataplane 145
11.3.2 Control plane 146
11.3.3 Client and server applications 147
11.4 Evaluation 148
11.5 Discussion and Future Work 150
11.6 Related Work 151
11.7 Chapter Summary 152

12 HovercRaft: Scalability and Fault-tolerance for μs-scale Services 153
12.1 Introduction 153
12.2 Motivation 155
12.3 Design 156
12.3.1 SMR-aware RPC layer 156
12.3.2 Separating RPC Replication from Ordering 157
12.3.3 Load balancing replies 158

12.3.4 Bounded Queues . . . 159
12.3.5 Load balancing Read-only Operations . . . 160
12.3.6 Join-Bounded-Shortest-Queue . . . 160
12.3.7 Communication in HovercRaft . . . 161
12.4 HovercRaft++ . . . 161
12.5 HovercRaft vs Raft . . . 162
12.6 Implementation . . . 164
12.6.1 R2P2 protocol extensions . . . 165
12.6.2 Raft extensions . . . 165
12.6.3 Multicast Flow Control and Recovery . . . 165
12.6.4 Aggregator implementation . . . 166
12.7 Evaluation . . . 168
12.7.1 One million SMR operations per second . . . 169
12.7.2 Scaling Cluster Sizes Without Regret . . . 170
12.7.3 Scaling to Improve Performance . . . 171
12.7.4 Loadbalancing in the presence of failures . . . 171
12.7.5 YCSB-E on Redis . . . 172
12.8 Discussion . . . 174
12.9 Related work . . . 175
12.10 Chapter Summary . . . 176

IV Wrapping Up **177**

13 Future Directions **179**
13.1 Latency vs Flow Completion Time . . . 179
13.2 Looking Up the Stack . . . 180
13.3 Distrusting the Datacenter . . . 180

14 Conclusion **183**

Bibliography **185**

Curriculum Vitae **215**

List of Figures

2.1 Basic communication pattern to execute a client request in a fault-tolerant group of 3 servers. Solid arrows refer to messages from and to the client based on the service-specific API, while dotted arrows refer to SMR messages. . . . 19

3.1 Comparison between a 16xM/D/1/FCFS and an M/D/16/FCFS queueing model. 24
3.2 Combining and comparing queueing models with synthetic microbechmarks 26

4.1 99th percentile latency for memcached USR measure with Mutilate with 144 and 320 connections with 1 outstanding request per connection. 32
4.2 Lancet's architecture depicting a coordinator (C), throughput agents (TA), latency agents (LA), symmetric agents (SA), and the target server under test. The dashed arrows correspond to the LANCET API while the solid ones are application RPCs 36
4.3 Lancet agent's state transition. Arrows represent messages from the coordinator. 37
4.4 LANCET's experiment methodology implemented by the coordinator. Dark grey boxes correspond to messages from the coordinator to the agents. Light grey boxes show the experiment end. 37
4.5 The structure of a LANCET agent. The grey part corresponds to the Python control plane, while the white part corresponds to the C dataplane communicating over shared memory (SHM). 40
4.6 Latency ECDF for an M/D/1 model and three deterministic workloads at 20% load. 42
4.7 Dynamics of LANCET's self-correcting methodology based on load for three service time distributions with $\bar{S} = 10\mu s$ 44
4.8 Impact of the inter-arrival distribution to tail-latency for a fixed with $\bar{S} = 10\mu s$. Vertical lines correspond to the load levels where Lancet reports inter-arrival distribution violations 45
4.9 Latency vs throughput graphs for a 5-client experiment with average service time of $\bar{S} = 1$ and $\bar{S} = 10$ 45

5.1 Approximation of a 2MPPS Poisson process with different packet sizes observed at 1/8 target servers 51
5.2 The SLOG dataplane design 53

6.1 Queuing models for $n = 2$ processors. 60
6.2 Simulation results for the 99th percentile tail latency for four service time distributions with $\bar{S} = 1$. 61
6.3 Maximum load that meets the SLO as a function of the mean service time $\bar{S}$. The SLO is set at $\leq (10 \times \bar{S})$ at the 99th percentile. The grey lines correspond to the ideal upper bounds determined by the centralized-FCFS and partitioned-FCFS models. 62
6.4 Dataflow in the ZYGOS operating system. Steps (1) – (6) correspond to the normal execution on the home core. Steps (a)-(b) occur during stealing and involve the home and remote cores. 65
6.5 Connection state machine transitions for the general case where an event is executed on a remote core (in blue). The connection is present in the shuffle queue exactly once when it is in the "ready" state, and never otherwise. . . . 69
6.6 99th percentile tail latency according to throughput for three distributions, each with 10µs and 25µs mean task granularity. The horizontal line corresponds to the SLO of $\leq 10 \times \bar{S}$. 72
6.7 Maximum load that meets the SLO of the 99th percentile $\leq 10 \times \bar{S}$. The grey lines correspond to the ideal upper bounds of the two theoretical, zero-overheads, models (centralized-FCFS and partitioned-FCFS). 72
6.8 Normalized rate of stealing vs throughput for exponential service time with mean 25 µs 73
6.9 99th percentile tail latency vs. throughput for two memcached workloads for Linux, IX and, ZYGOS. 74
6.10 Silo running the TPC-C benchmark. 74
6.11 Comparison of IX (batch size 1 and 64) and ZYGOS for a deterministic service time of 10 µs and 2 different SLOs. 77
6.12 Comparison between the FCFS and PS policies with policies that take advantage of application-level information: Earliest-Deadline-First(EDF), Shortest-Job-First(SJF), and Cycle Stealing Central Queue (CSCQ) 79

7.1 Sample 2-tier cloud application. Web servers handle web traffic coming from users' browsers and act as clients for the back-end servers that run the application logic and communicate with a managed database. The light green octagon is an external load balancer while the dark green one is an internal load balancer. 83
7.2 Connection-Request-Response (CRR) and Request-Response RR latency benchmarks on Azure with accelerated networking with and without an Azure internal load balancer 85
7.3 Commonly used load balancing schemes for cloud services based on VMs or containers. 86
7.4 Simulation result for tail latency to compare centralized and distributed policies. 90

7.5 A load balanced TCP handshake with and without connection redirection. Blue boxes correspond to IP headers, red boxes correspond to TCP headers. 91
7.6 Load Balancing over CRAB using the `Connection Redirect` TCP option. Dashed lines indicate connection establishment. Solid lines indicate data exchange. 93
7.7 Unloaded tail latency latency measured for the CRR and RR benchmarks for each CRAB load balancer implementation. 95
7.8 Maximum throughput (new connections per second) achieved by each CRAB load balancer implementation. 96
7.9 Unloaded tail latency for the CRR and RR benchmarks for setups with no load balancer (Direct), a DPDK-based L4 load balancer with DSR, and the CRAB implementation on DPDK. 97
7.10 Comparing Max goodput achieved in the CRR (dashed) and RR (solid) benchmarks for CRAB and a DPDK based L4 LB with DSR. I/O capacity of both LBs = 10G. I/O capacity of servers = 30G. 97
7.11 Load Balancing 48 single-core servers running a synthetic service time application with $\bar{S}$ = 1ms 99
7.12 Load Balancing 48 NGINX servers serving an 8 kB static file. 99

8.1 Replicated RPC experiment setup 106
8.2 One-way Flow Control FeedBack Loop. C_i indicates a connection, dark grey indicates occupied buffer space, light grey indicates the sliding window, B is the available receive buffer space. 108
8.3 99th percentile tail latency versus throughput for the three service time distributions with 10µs mean service time and the two flow control mechanisms, TCP and the proposed latency-aware (LA-fc). 112
8.4 Drop rate as a function of the offered throughput in the case of LA-fc for the 3 service time distributions. 112
8.5 GroupA (gA) throughput against server1 (s1) and server2 (s2), and 99th percentile tail latency for the two flow control mchanisms, TCP and the proposed latency-aware (LA-fc) 114

9.1 The R2P2 protocol for a request-reply exchange. Each message is carried within a UDP packet. Single arrows represent a single packet whereas double arrows represent a stream of datagrams. 120
9.2 R2P2 Header Format 122

10.1 Simulation results for the 99th percentile latency across 3 service time distributions with $\overline{S} = 1$ 129
10.2 Achieved Goodput as a function of the response size for the `JBSQ` policy on the software router managing 4 servers connected with 10GbE NICs compared to NGINX configured as HTTP reverse proxy loadbalancing the same 4 servers using a JSQ policy. 134

10.3 Tail-latency as a function of throughput for a zero service time synthetic RPC service for the software (SW-JBSQ) and the Tofino (P4-JBSQ) implementation of JBSQ compared to DIRECT. In DIRECT clients bypass the router and talk directly to servers by making a random server choice. 135
10.4 Evaluation of different load balancing policies for an exponential service time workload with $\bar{S} = 25\mu s$. 135
10.5 Synthetic Service Time Microbenchmarks. Service time $\bar{S} = 10\mu s$. 137
10.6 Bimodal service time with $\bar{S} = 10\mu s$ and 64 workers with single and multi-packet requests. DIRECT-1464 corresponds to an 1-packet request, while DIRECT-1465 and P4-JBSQ(5) correspond to 2-packet requests. 138
10.7 Comparison of R2P2 with the ZygOS [299] work-conserving scheduler: Exponential workload with $\bar{S} = 10\mu s$. 138
10.8 Lucene++ running on 16 16-threaded workers 139
10.9 99th percentile latency vs. throughput for Redis in a 4-node master/slave configuration. 139

11.1 SVEN's dataplane sample implementation running in Tofino's ingress pipeline. 145
11.2 99-th Latency for different service time distributions with $\bar{S} = 10\mu s$ without SVEN. 148
11.3 Throughput (MRPS) and 99-th Latency (µs) over time for a step load pattern and the same 3 service time distributions. The difference between the orange and the blue throughput lines corresponds to SVEN's drop rate. The red line shows the latency SLO@300µs. 149

12.1 Eliminating bottlenecks of SRM. Illustration on a 3-node cluster. 153
12.2 HovercRaft proposal in terms of layering compared to existing approaches. . 156
12.3 A view of the leader log. Each log entry includes the term, the designated replier, a pointer to the client request, and whether it is read-only. The replier is set only for the entries up until the announced_idx. 159
12.4 The communication patterns in our design resulting from separating replication from ordering and using in-network fan-in/fan-out. Solid arrows refer to application requests and replies. Dotted arrows refer to SMR messages. . . . 160
12.5 The in-network aggregator pipeline handling append_entries requests and replies 166
12.6 Tail latency vs. throughput for a fixed service time $S = 1\mu s$ workload with 24-byte requests and 8-byte replies on N=3 node cluster. 169
12.7 Achieved throughput under a 500µs SLO for a fixed service time service workload with $S = 1\mu s$, 8-byte replies and different request sizes. 170
12.8 Achieved throughput under a 500µs SLO for a workload with fixed service time of $S = 1\mu s$, 24-byte requests and 8-byte replies for different cluster sizes. 170
12.9 Latency versus throughput for $S = 1\mu s$ fixed service time, 24-byte requests, and 6kB replies for different cluster sizes. 172

12.10 Latency versus throughput for a $S = 10\mu s$ service time with bimodal distribution on a 3-node cluster. Requests are 24-bytes, replies are 8-bytes, and 75% of operations are read-only. 172
12.11 99-th percentile latency and throughput as a function of time in the presence of failures a leader failure for a HovercRaft++ 3-node cluster and a bimodal distribution of $\bar{S} = 10\mu s$ with 75% read-only operations. 173
12.12 Latency vs throughput for YCSB-E(95% SCAN 5% INSERT) on Redis with our custom module to support YCSBE-E operations. 174

List of Tables

4.1 Classification of concerns related to running a latency experiment into workload, methodology, and measuring tool-specific components. We advocate that the Workload column has to be user defined, while Methodology and Measuring Tool columns have to be handled systematically by the measuring framework. 33

4.2 The LANCET coordinator API with the information returned by the agents on each call. For the `get_throughput` and `get_latency` requests, the agents also reply information related to the Inter-Arrival distribution (IA), the latency Confidence Intervals (CI), and whether the collected samples are stationary and *iid*. If they are not *iid*, the reply contains the target sampling rate necessary to get *iid* data. 36

5.1 Challenges and how we approach them in SLOG 52

6.1 Maximum throughput under the SLO of 1000 µs and respective latencies at approximately 50%, 75%, and 90% of that load for each Silo running the TPC-C benchmark. Load is in KTPS. The number in the parentheses is the ratio of the 99th percentile end-to-end latency to Silo's 99th percentile service time (203µs). 76

7.1 Feature comparison between different deployed load balancing schemes and CRAB. 88

9.1 The R2P2 message types 122

9.2 The `r2p2-lib` API 124

12.1 Comparison of Rx and Tx message overheads for the leader in Raft, HovercRaft, and HovercRaft++ for the non-failure case. N is the number of nodes. 162

1 Introduction

Online services play a major role in our everyday life for communication, entertainment, socializing, e-commerce, *etc.*. The COVID-19 global pandemic [279] and social distancing has made those services even more fundamental [72]. There are more than 4.5 billion people [160] (almost 60% of the global population) that are active internet users making web-searches, exchanging emails, watching videos, and buying products online. Google serves more than 3.5 billion queries per day [134]. Facebook has more than 1.5 billion active users [110]. There are more that 500 hours of video uploaded on YouTube every minute [373] and more than 300 billion emails exchanged daily [97]. Amazon ships over 3 million packages a day, while the same number for Alibaba is 12 million [318].

These online services have to support the global massive scale while being interactive. Usability studies have shown that users are sensitive to hundreds of milliseconds delays [270, 172]. Such delays affect users' engagement and translate to revenue loss. A delay in the order of 100 to 400 milliseconds in Google's search results reduces the number of searches per user from of 0.2% to 0.6% on average [322]. A similar A/B testing experiment showed that users preferred a web-search to return 10 results in 0.4 seconds rather than 30 results in 0.9 seconds. Half a second delay caused 20% drop in traffic despite the improved quality of search [213]. Similarly, Amazon reports that every 100ms delay causes 1% drop in sales [212]. Thus, online services have to perform under strict latency requirements, described as Service-Level-Objectives (SLO) and expressed as the latency for some tail percentile of the incoming requests, *e.g.*, 500µs at the 99-th percentile.

The heart of online services is the datacenter. Datacenters can be seen as the evolution of mainframes. Driven by the needs for high-throughput and fault-tolerance and the economy of scale, datacenters follow a scale-out paradigm and comprise of cheap commodity hardware instead of expensive scale-up installments. The design, construction, and operation of a datacenter is completely controlled by the owning company, thus allowing several degrees of freedom in terms of customization according to the company needs. Unlike the public internet where backwards compatibility and compatibility among different vendors is absolutely necessary, the datacenter is a field of innovation and experimentation across the software and

hardware stack.

In this thesis, we focus on latency-critical systems running inside a single datacenter. We carefully study their interactions, the causes that cause long tail latencies, and propose solutions that improve µs-scale latencies by taking advantage of the freedom to deploy new software and hardware mechanisms and novel datacenter-specific abstractions.

1.1 RPCs in the Epicenter

The basis for most intra-datacenter and inter-datacenter communications is *Remote Procedure Calls* (RPCs). The idea of RPCs dates back to the early days of distributed computing [38] and initially described code execution in the context of a remote machine. Today the term is used to described any generic interaction between a client and a server machine. Protocols used every day, such as `HTTP`, follow this scheme, while frameworks, such as gRPC [139], stay closer to the initial definition. In this thesis we will focus in intra-datacenter RPCs.

The existing hardware infrastructure used in modern datacenter is capable of communication latencies in the scale of µs [31]. On top of this infrastructure applications, such as key-value stores, serve data directly off memory to reduce service time, leading to service times that are handful of µs. Thus, the end to end communication latency for datacenter RPCs has the potential to be in the scale of µs. Existing operating systems and operating system mechanisms, *e.g.,* scheduling, were designed with an entirely different set of assumption during an era when IO was more expensive in the ms-scale. Barroso et al. [31] quantify the overhead of the existing software layers on top of modern hardware.

Challenge 1: Redesign existing operating systems for µs-scale computing.

TCP has emerged as the main transport protocol both for inter and intra-datacenter communications. Thus, RPCs are commonly delivered on top of TCPs reliable bytestream abstraction. Despite the layering benefits and the separation of concerns, the abstraction mismatch between RPCs (request-response) and TCP (bidirectional bytestream) leads to several inefficiencies and design limitations. These limitations affect both the RPC latency, *e.g.,* the problem of head of line blocking at the server, and the efficiency of server resources, *e.g.,* the problem of bufferbloat at the servers due to increasing number of connections. The backwards compatibility concerns limits the deployment of new protocols in the public internet. A datacenter, though, does not suffer from such limitations.

Challenge 2: Design new transport protocols specifically for µs-scale datacenter RPCs.

1.2 The Need for Tail-Tolerant Systems

The scale-out design of modern datacenters is not limited to hardware. It extends to software and the way datacenter applications are deployed. Serving a client request might involve

several datacenter machines. This high fan-out communication pattern in combination with the inherent application service time variability leads to the problem called *tail at scale* [80]. The end to end latency of a client request is defined by the slowest sub-request it triggers inside the datacenter. Large fan-out patterns increase the probability of including a slow sub-request in almost every client request. Thus, tail phenomena become commonplace as a result of the law of large numbers.

Mitigating long tail latency requires understanding its sources first. According to Dean et al. [80] long tail latency might be the result of interference, garbage collection, background tasks, *etc.*. Those problems can be tackled with resource isolation [231] and careful system tuning [223]. Another reason behind long tails is persistent or transient load imbalance due to poor load balancing or scheduling. Despite queuing theory having a clear answer on the tail-optimal scheduling policy [366] (there is no optimal scheduling policy; *First-Come-First-Serve* is optimal for low service time variability; *Processor-Sharing* is optimal of high service time variability), existing state of the art datacenter operating systems, *e.g.*, IX [33], operate with multiple FCFS queues independently of the application.

Challenge 3: Design new tail-tolerant scheduling and load-balancing mechanisms for µs-scale RPCs that eliminate latency SLO violations and avoid IO bottlenecks.

1.3 The Advent of In-Network Compute

The end of Moore's law and the ever increasing need for performance and efficiency has led researchers and datacenter operators to designing and deploying reconfigurable [53] or custom architectures [173] for datacenter workloads. This trend also affects the datacenter networking infrastructure with the advent of programmable switches [28, 44, 54] and NICs [241, 265, 246]. These devices offer certain level of programmability, while still running at line rate, thus posing an interesting challenge on whether it makes sense and how to offload functionality commonly performed at the end-host to the network.

Opportunity: Leverage in-network programmability to accelerate datacenter applications.

1.4 Thesis Statement

Above we identified three challenges and one opportunity related to latency-critical datacenter systems. The top goal of this thesis is to drastically improve the tail latency and tail-tolerance of µs-scale RPCs in a scale-out datacenter environment. These techniques span across operating systems and networking protocols, while they aggressively push functionality to the network leveraging in-network compute. A second-level goal of this thesis is performance reasoning and systematic evaluation through robust and reusable experiment methodologies and benchmarking tools.

We, first, propose new latency measuring tools that are specifically designed for latency experiments in a µs scale and an evaluation methodology that quantifies the system overheads. Having the correct arsenal of tools, we focus on existing abstractions, specifically TCP-based datacenter communications, and we provide solutions for intranode RPC scheduling and cloud workload loadbalancing. We also extend existing abstractions, without breaking compatibility, with new features for SLO-aware tail-tolerance. Then, given the datacenter freedom, we introduce new abstractions that are better fitted to the RPCs semantics. We show how this change improves existing mechanisms, such as load balancing, while it enables us to push new functionality, *e.g.,* fault-tolerance, to the network layer.

Thesis Statement

The datacenter is a dynamic environment in which scale-out applications need to run under strict µs-scale tail latency SLOs. Operating under a single managing authority enables and encourages the deployment of customized hardware and software infrastructure. Co-designing network and operating system mechanisms, pushing server functionality to the network, and systematically approximating optimal scheduling policies significantly improves tail latency in µs-scale RPC systems, while enabling tail and fault-tolerance guarantees.

1.5 Thesis Contributions

Specifically, this thesis makes the following contributions:

1. **A novel comparative methodology for system evaluation**

   We propose a system evaluation methodology that depends on synthetic microbenchamarks and system modelling with queueing theory. Our modelling with queueing theory enables fast design exploration and bottleneck analysis. The use of synthetic microbenmarks with configurable service times and request/reply sizes enables matching the theoretic queueing results with the actual system implementation to accurately quantify the system and communication overheads. Such an analysis is crucial especially when building systems for µs-computing.

2. **Tools and a robust methodology to measure latency accurately**

   We introduce LANCET, a self-correcting tool designed to measure the open-loop tail latency of µs-scale datacenter applications with high fan-in connection patterns. LANCET is self-correcting as it relies on online statistical tests to determine situations in which tail latency cannot be accurately measured from a statistical perspective. The workload configuration, the client infrastructure, or the application itself could, under circumstances, prevent accurate measurement. Because of its design, LANCET is also extremely easy to use. In fact, the user is only responsible for (i) configuring the workload parameters, *i.e.,* the mix of requests and the size of the client connection pool, and (ii)

setting the desired confidence interval for a particular tail latency percentile. All other parameters, including the length of the warmup phase, the measurement duration, and the sampling rate, are dynamically determined by the LANCET experiment coordinator. When available, LANCET leverages NIC-based hardware timestamping to measure RPC end-to-end latency. Otherwise, it uses an asymmetric setup with a latency-agent that leverages busy-polling system calls to reduce the client bias.

Our evaluation shows that LANCET automatically identifies situations in which tail latency cannot be determined and accurately reports the latency distribution of workloads with single-digit µs service time. For the workloads studied, LANCET can successfully report, with 95% confidence, the 99th percentile tail latency within an interval of ≤ 10µs. In comparison with state-of-the-art tools such as Mutilate and Treadmill, LANCET reports a latency cumulative distribution that is ~20µs lower when the NIC timestamping capability is available and ~10µs lower when it is not.

We, also, introduce SLOG, a programmable load generator and latency-measuring tool based on a programmable switch. SLOG repurposes programmable switches and enables us to use them as a tool for latency and throughput experiments. It leverages the programming capabilities and the fixed function units of a Tofino ASIC to generate load and measure tail latency for both NFs and RPC services. Through a novel dataplane implementation SLOG is able to generate a Poisson arrival distribution despite the hardware limitations. According to our knowledge, SLOG is the only hardware-based tool that is able to generate a randomized inter-arrival distribution, necessary for a realistic latency experiment.

3. **A tail-optimal scheduler for µs-scale tasks with moderate variability**

   We present ZYGOS, a system optimized for µs-scale, in-memory computing on multicore servers. ZYGOS implements a work-conserving scheduler within a specialized operating system designed for high request rates and a large number of network connections. It uses a combination of shared-memory data structures, multi-queue NICs, and inter-processor interrupts to rebalance work across cores.

   For an aggressive service-level objective expressed at the 99th percentile, ZYGOS achieves 75%of the maximum possible load determined by a theoretical, zero-overhead model (centralized queueing with FCFS) for 10µs tasks, and 88%for 25µs tasks. We evaluate ZYGOS with a networked version of Silo, a state-of-the-art in-memory transactional database, running TPC-C. For a service-level objective of 1000µs latency at the 99th percentile, ZYGOS can deliver a 1.63×speedup over Linux (because of its dataplane architecture) and a 1.26×speedup over IX, a state-of-the-art dataplane (because of its work-conserving scheduler).

4. **An L4 load balancing scheme that bypasses the load balancer on the datapath**

   We propose CRAB, an alternative L4 load balancing scheme that eliminates latency overheads and scalability bottlenecks while simultaneously enabling the deployment of complex, stateful load balancing policies. A CRAB load balancer only participates

in the TCP connection establishment phase and stays off the connection's datapath; thus provisioning load balancer depends on the rate of new connections rather than the actual connection bandwidth. CRAB depends on a new TCP option that enables connection redirection. We provide different implementations for a CRAB load balancer on different technologies, *e.g.,* DPDK and eBPF, showing that a CRAB load balancer does not require many resources to perform well. We introduce the connection redirection option to the Linux kernel with minor modifications, so that it that can be shipped with the VM images offered by the cloud providers. We show how the same functionality can be achieved with a vanilla Linux kernel using a Netfilter module, while we discuss how CRAB can work while clients and servers remain completely agnostic, based on functionality added on the host.

Our evaluation shows that CRAB pushes the IO bottleneck from the load balancer to the servers in cases where vanilla L4 load balancing does not scale and provides end-to-end latencies that are close to direct communication while retaining all the scheduling benefits of stateful L4 load balancing.

5. **A novel transport protocol for datacenter RPCs that enables in-network policy enforcement**

   We identify the mismatch between streaming or datagram-based transport protocols and RPCs that imposes overheads and limits the design flexibility. We propose R2P2, a UDP-based transport protocol specifically designed for RPCs inside a datacenter. Our work exposes the RPC abstraction to the endpoints and the network, making RPCs first-class datacenter citizens. R2P2 is specifically designed for efficient in-network policy enforcement by separating the policy enforcing mechanism from request and reply streaming. Any RPC policy logic can be implemented either in a software middlebox or within a P4 switch ASIC pipeline. By exposing the right abstraction and making the network RPC-aware R2P2 is the catalyst that enables pushing server-side functionality that is common among applications in the transport layer and the network. We use the new protocol to implement several in-network policies such as RPC load balancing, flow control, and fault-tolerance.

6. **The Join-Bounded-Shortest-Queue RPC scheduling policy**

   Join-Bounded-Shortest-Queue (JBSQ) is a novel split-queue scheduling policy which exposes the trade-off between optimal scheduling and high throughput. We implement `JBSQ` as an in-network RPC policy for request level load balancing. `JBSQ` lowers tail latency by centralizing pending RPCs in the router and ensures that requests are only routed to servers with a bounded number of outstanding requests.

   Our evaluation, using a range of microbenchmarks, shows that `JBSQ` on R2P2 is suitable for µs-scale RPCs and that its tail latency outperforms both random selection and classic `HTTP` reverse proxies. The P4-based implementation of JBSQ on a Tofino ASIC adds less than 1µs of latency whereas the software middlebox implementation adds 5µs latency and requires only two CPU cores to route RPCs at 10 Gbps line-rate. R2P2

with JBSQ improves the tail latency of web index searching on a cluster of 16 workers operating at 50% of capacity by 5.7× over NGINX. Also, it improves the throughput of the Redis key-value store on a 4-node cluster with master/slave replication for a tail-latency service-level objective of 200µs by more than 4.8× vs. vanilla Redis.

7. **The notion of SLO-aware flow control**

   Tail-latency tolerance (or just simply tail-tolerance) is the ability for a system to deliver a response with low-latency nearly all the time. It it typically expressed as a system metric (*e.g.,* the 99th or 99.99th percentile latency) or as a service-level objective (*e.g.,* the maximum throughput so that the tail latency is below a desired threshold).

   We advocate instead that modern datacenter systems should incorporate tail-tolerance as a core systems design principle and not a metric to be observed, and that tail-tolerant systems can be built out of large and complex applications whose individual components may suffer from latency deviations. This is analogous to fault-tolerance, where a fault-tolerant system can be built out of unreliable components.

   The general solution is for the system to control the applied load and keep it under the threshold that violates the latency SLO. We implement such an SLO-aware flow control mechanism on top of TCP and R2P2. For TCP we maintain a wire-compatible header format without introducing extra messages. We implement a proof-of-concept userspace TCP stack on top of DPDK and we show that the new flow control mechanism prevents applications from violating service-level objectives in a single-server environment by throttling the incoming requests. We demonstrate the true benefit of the approach in a replicated, multi-server scenario, where independent clients leverage the flow-control signal to avoid directing requests to the overloaded servers. For R2P2, we propose to augment RPC semantics with an architectural layer at the R2P2 middlebox that measures the observed tail latency and probabilistically rejects RPC requests maintaining throughput under the threshold that violates the SLO. Our design is application-independent, and does not make any assumptions about the request service time distribution.

8. **A scalable in-network accelerated protocol for fault-tolerant datacenter RPCs**

   We propose HovercRaft, a new way to build generic fault-tolerant RPC services by providing fault-tolerance at the RPC layer, leveraging the exposed RPC semantics offered by R2P2. HovercRaft deals with the replication trade-off according to which adding nodes to a system can either increase performance at the expense of consistency, or increase resiliency at the expense of performance. We introduce an approach by which adding nodes increases both the resilience and the performance of general-purpose state-machine replication. We achieve this through an extension of the Raft protocol that carefully eliminates CPU and I/O bottlenecks and load balances requests.

   Our implementation uses state-of-the-art kernel-bypass techniques, datacenter transport protocols, and in-network programmability to deliver up to 1 million operations/second for clusters of up to 9 nodes, linear speedup over unreplicated configuration for

selected workloads, and a 4× speedup for the YCSBE-E benchmark running on Redis over an unreplicated deployment.

1.6 Thesis Organisation

This thesis is organised as follows. In Chapter 2 we go through the necessary background on datacenter systems and mechanisms that we modify or depend on in the rest of the thesis. The rest of the thesis is organised in 4 parts.

1. **Part 1** summarises the tools and methodologies developed and used in the course of this thesis to measure latency and evaluate systems. Chapter 3 describes our proposed evaluation methodology on synthetic microbechmarks and modelling. Chapter 4 introduces LANCET, a self-correcting latency measuring tool that uses hardware timestamping and its robust experiment methodology on statistical testing. Chapter 5 presents SLOG a load generator on a programmable switch that takes advantage of the hardware implementation to achieve high throughput and measuring accuracy.

2. **Part 2** includes the scheduling and load balancing contributions of the thesis for latency-critical systems on existing abstractions, namely TCP's bytestream. Chapter 6 describes the ZYGOS operating system and its work-conserving scheduler for µs-scale RPCs on top of TCP. Chapter 7 presents a L4 load balancer for internal cloud workloads that depends on connection redirection feature introduced to TCP that removes the load balancer from the data path, without breaking backwards compatibility. Finally, Chapter 8 introduces the proposed SLO-aware flow control mechanism on top of TCP that eliminates SLO violations and can be used to improve tail latency in a real deployment scenario with several replicas.

3. **Part 3** deals with the inefficiencies and limitations due to the abstraction mismatch between RPCs and TCP by introducing a new transport protocol for RPCs and describes how the new protocol enables pushing existing and new functionality in the network. Chapter 9 presents the R2P2 protocol and describes the mechanism for in-network policy enforcement. Chapter 10 introduces the JBSQ policy to perform request-level load balancing. Chapter 11 describes SVEN a system that implements in-network SLO-aware RPC flow control and provides similar guarantees to TCP's equivalent mechanism (Chapter 8) without requiring modifications to the networking stack. Finally, Chapter 12 identifies the potential of pushing new functionality to the transport layer and describes HovercRaft, a system that integrates a state machine replication algorithm in R2P2 and uses in-network compute to accelerate fault-tolerant RPCs.

4. **Part 4** describes future research directions and concludes the thesis.

1.6.1 Bibliographic Notes

This thesis was conducted under the supervision of my advisor Ed Bugnion and parts of it have been published with the help of my colleagues. Chapter 4 was published in the 2019 Usenix Annual Technical Conference (ATC 2019) [201]. Chapter 5 was presented as a poster in the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2020) [203]. Chapter 6 was published in the 26th ACM Symposium on Operating Systems Principles (SOSP 2017) [299]. Chapter 7 was published in the ACM Symposium on Cloud Computing 2020 (SoCC '20) [200]. Chapter 8 was published in the 2018 Workshop on Kernel Bypassing Networks (KBNets 2018) at SIGCOMM 2018 [197]. Chapter 9 and Chapter 10 come from a paper presented in the 2019 Usenix Annual Technical Conference (ATC 2019) [202]. Chapter 11 was presented in the 4th Asia-Pacific Workshop on Networking (APNet 2020) [199]. Finally, Chapter 12 was presented in Eurosys 2020 and was awarded with the Best Student Paper Award [198].

Note that ZYGOS was co-authored with George Prekas. This thesis does not include the contributions of Subsection 6.4.5.

2 Background and Challenges

In this chapter we describe the specific characteristics of the datacenter environment that motivate and enable the proposed system designs in the next chapters. We discuss why existing system and networking mechanisms fail to meet the requirements for µs-scale datacenter services and how the unique datacenter characteristics affect the entire software and hardware stack enabling specialized solutions.

2.1 Datacenter Infrastructure and Applications

The datacenter environment provides a challenging opportunity for system researchers because of a unique combination of physical characteristics, ease of new solution deployment, and performance requirements. A datacenter is a physically isolated building with commodity storage, network, and compute infrastructure that runs under the administrative domain of one entity, namely the owning company. The administrative entity can fully control every aspect of the datacenter from the building construction, power management, and cooling, all the way up to the compute, network, and storage hardware, network topology, and cabling. Despite some level of heterogeneity due to new hardware roll-out and hardware failures, a datacenter can be seen as a homogeneous infrastructure, compared for example with the domain of mobile computing in which the device capabilities and the quality of connectivity can vary significantly. In many cases, depending on the business model, the administrative entity also fully controls the software running on this infrastructure. For example, companies such as Facebook run only company-related workloads. This combination allows for easy and aggressive deployment of custom software across the stack that may break compatibility with existing protocols and APIs. A step further in providing solutions specifically tailored to the application needs is software-hardware co-design that is becoming more and more popular in the datacenter today.

Next we focus on the hardware and software infrastructure running inside a datacenter. We do so to understand our target deployment environment and the current state of the art. The level of customization across the stack is a motivation for our proposed systems.

Network Topology: The first layer of innovation and customization inside a datacenter that is relevant to this thesis begins at the network topology. Different companies may deploy different network topologies [320, 307] in their datacenters. All these topologies, though, share some common characteristics that are significantly different from the public internet. The use of commodity hardware and the constantly increasing need for high bisection bandwidth in Pbit/s has led datacenter designers to Clos networks, usually organised in three tiers. The two aspects that characterize a datacenter topology is multipathing and oversubscription. A Clos topology depends on redundant paths to increase capacity, while the upper tiers are oversubscribed to reduce the total cost of ownership.

Currently most datacenters deploy static topologies that do not change unless there are link failure or re-cabling due to maintenance. Research efforts, though, in switching hardware enables dynamic topology reconfigurations through optical switching. Datacenter researchers are now exploring the opportunities of adapting the network topology to the workload needs and dynamically change it as the workload evolves [248, 129]. Such approaches provide several challenges for network management, routing, and application APIs, while still suffer from the long reconfiguration latencies in the µs-scale. However, they provide an excellent example and usecase of how new technology can be deployed fast in the datacenter and change the way the datacenter operates, while introducing significant performance gains.

Datacenter Hardware for Networking: Since their early days datacenters have been defined been defined by commodity technologies due to their cost benefits and the economies of scale. This leads to scale-out architectures that depend on cheap and redundant components that can fail. By looking at the Open Compute Project [278] that describes commonly used datacenter equipment, one can observe that this trend characterizes the storage, compute, and networking components of a datacenter. From a networking perspective, datacenters deploy cut-through switches with shallow buffers that have a few hundreds of nanoseconds of switching latency. Servers are connected to the Top Of the Rack switch (ToR) with 10G or 40G links using commodity Ethernet [320] or RoCE (RDMA over Converged Ethernet) [140], while new devices already support even higher network bandwidth, *e.g.,* at 400G [12, 63]. Examples of commonly used network interfaces are the Mellanox ConnectX series [247], or Intel's 82580 [158] and 710[159] network adapters. Based on the deployed hardware and the underlying network topology we understand that datacenter round-trip times are just a few µs. The worst case for a packet is six network hops on top of a 10G link.

In-Network Programmability: The end of Moore's law and the need for better performance and efficiency has led researchers to enhance existing commodity-based infrastructures with custom domain-specific architectures. The datacenter is an ideal testbed for this experiment. In the networking domain this trend takes the form of in-network programmability. Programmable devices, such as programmable switches [28, 44, 54] and smartNICs [241, 265, 246], are becoming part of the networking path and undertake tasks traditionally performed on the server end. Depending on the device, programmability is achieved through the use of

FPGAs [246], multicore network processors [265], or programmable ASICs [28]. The underlying technology defines the programmable interface with the device. FPGA-based NICs require writing code in RTL languages [101]. Network processors can be programmed through vendor-specific APIs [156]. Programmable ASICs, such as Barefoot's Tofino, are programmed through a language specifically designed to describe network pipelines, called P4 [39]. P4 abstracts the underlying hardware and exposes a programmable interface that depends on match-action tables.

The advent of those new programmable devices extends the ways existing infrastructure is deployed. For example, Microsoft Azure offloads the virtual networking functionality on an FPGA-based smartNIC [116] and Facebook runs a L4 stateful load balancer on a P4 switch [251]. More aggressive approaches to in-network compute push functionality that has typically run on servers to those network devices, *e.g.,* key-value stores implemented on programmable switches [171].

Datacenter Networking: The 3-tier clos topology, µs-scale RTTs, the introduction of programmable devices, and the single administrative domain drives the development of new network protocols specifically for the datacenter. Unlike the public internet, datacenter network control planes can be centralized [320] and must provision for multi-path routing. Deployed solutions for multi-path routing range from simple hash-based approaches, such as Equal-Cost Multi-Path routing (ECMP) [154], to more complicated ones [6, 130, 150, 190, 354, 380] that also try to equally load balance across the different paths. At the transport protocol level TCP is widely used based on the DCTCP [7] congestion control algorithm, while new congestion control algorithms have been proposed [9, 125, 143, 23, 288, 59, 255, 252] that take advantage of specific datacenter features. For example, low communication latencies enable precise delay-based congestion control [252] and receiver-driven congestion control [255, 143]. Also, in-network programmability allows switches to fully describe congestion levels [13], instead of using approximations based on ECN for example. Finally, the datacenter environment and the available hardware enables features and mechanisms across the networking protocol stack that could not be deployed in such a massive scale before, *e.g.,* IP multicasting [317] and source routing [225, 175].

Datacenter Operating Systems: The above hardware description shows that the datacenter is an environment where the hardware infrastructure can support µs-scale communication latencies. However, existing commodity software stacks are not designed for such timescales and the datacenter environment, resulting in two main problems. The first one is the added overhead due to inefficient software layers that was not a problem in the millisecond-scale. Serving minimum-sized packets on a single CPU for a 10G network means that each packet should be processed in less than a µs. Any added system overheads would result in underutilizing the hardware potential. Barroso et. al [31] show that while the datacenter fabric can support an RDMA operation in 2µs, the additional software layers result in a round-trip time around 75µs. The second problem is that existing infrastructure cannot deal with µs-scale latencies,

unlike millisecond and nanosecond-scale ones. A hard drive read is a millisecond-scale operation and operating systems can hide this latency through context switching. Similarly, a cache access is a nanosecond-scale operation and modern architectures can hide this latency through out of order execution. Thus existing operating systems need to be re-designed for µs-scale computing.

The proposed datacenter operating systems eliminate most software overheads by bypassing the traditional kernel stacks. These technologies include kernel-bypass development kits [88, 306], user-level networking stacks [167, 277, 239, 316, 167, 291, 331], stacks that bypass the POSIX layer [142], protected dataplanes [34, 290], and microkernel-based approaches [191, 240]. These approaches bypass the kernel and rely on I/O polling to both increase throughput and reduce latency jitter [219, 223] coming from interrupt handling. Also, such approaches statically partition resources among different cores to eliminate inter-core communication.

However, the performance benefits of the above approaches come through sweeping simplifications and by eliminating functionality that operating systems used to perform, *e.g.,* software-based scheduling. For example, IX [34] a state-of-the-art dataplane operating system for the datacenter uses RSS [308] to split incoming packets across different cores. Then, its synchronization-free nature forces each thread to process *only* the packets that were directed to it by the NIC hardware. Such a design, despite improving throughput, has a dramatic impact on tail latency when the load is below saturation as some cores may be idle while others have a backlog of packets to be processed. One of the challenges this thesis deals with is how to combine the performance benefits of dataplane operating systems with more elaborate and work-conserving scheduling policies present in commodity operating systems, but removed for performance.

Datacenter Applications and the Tail at Scale Problem: We now describe the types of applications that run on top the above versatile and dynamic software and hardware infrastructure. Inside a datacenter there are several different application classes with different resource requirements that serve different purposes for the owning company. These applications might range from batch jobs, such as big data analytics and machine learning training, that have high CPU and network requirements but are not latency-critical, to latency-critical applications with µs-scale service times that need to perform under strict Service Level Objectives (SLOs) for their tail latency because they are user facing. In this thesis we will focus on the latter.

These latency-critical applications include in-memory key-value stores, such as Memcached [250] and Redis [304], in-memory databases such as Silo [337], and web-search, such as Apache Lucene [233]. The deployment of these applications follows the datacenter scale-out design, in which there are different instances of the same application serving a different parts of the dataset. Consequently, serving a client request to an online service, such as web-search, requires many different machines inside the datacenter. For example, loading a Facebook page requires on average more than 500 of those sub requests [271]. This communication pattern in which the initial request breaks down to several sub-requests is called a fan-out/fan-in

pattern [80]. In such fan-out patterns the top-level request is handled by the *root* nodes that send sub-requests to *leaf* nodes. Leaves reply back to the root node and the root node replies back to the client.

Such applications that follow large fan-out/fan-in patterns suffer from the so-called *tail at scale* problem [80]. Unavoidably there is variability in the application service times that can or cannot be attributed to the application logic itself. This variability might come due to contention for shared resources, such as main memory, background daemons, power limits, garbage collection, interrupt serving, OS scheduling, *etc.*. So, the same request might take a different amount of time depending on the state of the server. As a result, as the fan-out size increases, the probability of hitting a slow request increases, too. The latency of the top request is defined by the slowest sub-request. A request that fans-out to 100 different requests is highly likely to always observe the 99-th percentile worst end-to-end latency due to the tail at scale problem.

There are different approaches to deal with the tail at scale problem, depending on the reasons behind service time variability. Solutions such as request hedging, namely sending the same request to different replicas and keeping the fastest reply, try to remain agnostic to the reasons behind service time variability. Solutions, such as Heracles [231] focus on interference for shared resources. In this thesis we will focus on how to deal with the tail at scale problem and latency violations that are the result of scheduling decisions and imbalance.

Public Cloud: Another datacenter use, apart from running applications required by the owning company, is the public cloud. Companies such Amazon, Microsoft, and Google offer their infrastructure for rent through cloud services. There are different flavours of the public cloud depending on the cloud tenant needs and how involved they want to be in the infrastructure management. These flavours range from bare metal machines [15] to function as a service [16]. Companies as big as Netflix [18] might choose the public cloud for their services. Consequently, the same challenges of offering high throughput and low latency services remain for cloud providers. However, they have to do so without being able to modify the application, without the freedom of introducing non-backwards compliant protocols, and while guaranteeing physical and performance isolation between tenants. In this thesis we will see how we can modify certain services offered by cloud providers leveraging the cloud deployment model in order to make them more efficient without breaking compatibility for cloud tenants.

2.2 Datacenter Remote Procedure Calls

Datacenter services, both external and internal, are typically exposed through remote procedure calls (RPC). TCP has emerged as the main transport protocol for latency-sensitive, intra-datacenter RPCs running on commodity hardware, as its reliable stream semantics provide a convenient abstraction to build upon. RPCs are sent on top of TCP's bytestream abstraction using protocols such as HTTP, serialization approaches such as Thrift [333] and

gRPC [139], or even application-specific binary formats, *e.g.,* Memcache's binary protocol.

The problem of efficiently handling incoming RPCs dates back to the original C10k problem [50] when socket scalability was the primary bottleneck. The initial approaches to building scalable applications allocated a kernel thread or process per connection; servicing a new request required a scheduling decision. However, despite the sophistication of modern operating system schedulers such as Completely Fair Scheduler (CFS) [55] and Borrowed Virtual Time (BVT) [93], context switch and stack management overheads made developers move to more performant designs to serve incoming requests.

Today, fine-tuned commodity operating systems can serve millions of requests per second and over a million of concurrent connections on a commodity server [365, 289, 371]. These scalable designs fall into two main event-oriented patterns: symmetrical and asymmetrical ones. Symmetrical designs split connections onto threads, and each thread interacts with the operating system using non-blocking system calls. This pattern is used by the popular `libevent` and `libuv` frameworks [226, 228]. On Linux, this pattern typically relies on the `epoll` system call, which has long provided a way to statically map connections to threads. To avoid cases of load imbalance across cores because of imbalance across connections, developers tried sharing the same connection among multiple threads. However, this led to thundering herd problems [206]. The recent addition in Linux 4.5 of `EPOLLEXCLUSIVE` avoids such problems since in most cases only one thread is woken up to serve `epoll` [100].

In the asymmetrical pattern, a small number of threads handle all network I/O, identify RPC boundaries and add RPC requests to a centralized queue from which other tasks pull requests. This pattern is used by frameworks such as `gRPC` [139] and applications such as recent versions of `nginx` [268] and Apache Lucene [233]. While this pattern may increase the latency of an individual request and impact throughput when the tasks are small, it provides for an elegant separation of concerns and enables the efficient use of all worker cores.

However, using TCP for RPC workloads is quite a deviation from the original design of a wide-area, connection-oriented protocol for both interactive (*e.g.,* `telnet`) and file transfer applications. TCP's generality comes with a certain cost as RPC workloads usually consist of short flows in each direction. In many cases, the requests and replies are small and can fit in a single packet [14, 255]. Although RDMA is an alternative, it has specific hardware requirements and can be cumbersome to program, leading to application-specific solutions [90, 181, 180].

Overall, the requirements of RPCs differ from the assumptions made by TCP in terms of failure semantics, connection multiplexing, API scalability, and end-point buffering:

RPC semantics: Some datacenter applications choose weak consistency models [81] to lower tail latency. These applications typically decompose the problem into a series of independent, often idempotent, RPCs with no specific ordering guarantees. Requests and responses always come in pairs that are semantically independent from other pairs. Thus, the reliable, in-order

stream provided by TCP far stronger than the applications needs and comes with additional network and system overheads.

Connection multiplexing: To amortize the setup cost of TCP flows, RPCs are typically layered on top of persistent connections, and most higher-level protocols support multiple outstanding RPCs on a flow, *e.g.,* `HTTP/2`, memcache, *etc.* Multiplexing different RPCs on the same flow implies ordering the requests that share a socket, even though the individual RPCs are semantically independent. This ordering limits scheduling choices and can lead to Head-of-Line-Blocking (HOL). HOL appears when fast requests are stuck behind a slower request and when a single packet drop affects multiple pending requests.

Connection scalability: The high fan-in, high fan-out patterns of datacenter applications lead to large number of connections and push commodity operating systems beyond their efficiency point. Recent work has addressed the issue either by deviating from the POSIX socket interface while maintaining TCP as the transport [34] or by developing custom protocols, *e.g.,* to deploy memcached on a combination of connection-less UDP for RPC `get` and router proxy for RPC `set` [271].

Endpoint bufferbloat: Prior work has addressed network-specific issues of congestion management and reliability within the network [7, 8]. Unfortunately, the use of TCP via the POSIX socket API leads to buffering in both endpoints over which applications have little control or visibility [197]. Applications willing to trade-off harvest vs. yield [120] would ideally never issue RPCs with no chance of returning by the deadline due to buffering in the network stack.

2.3 Flow Control

Flow control is one of the profound examples of Salzer's end-to-end argument [311]. Although the notion of flow control is better understood in the context a sender and receiver where it guarantees that the sender's sending rate matches the receiver capacity to process incoming data, it generalizes to other systems too. Flow control guarantees that a system has enough capacity to serve new incoming work and creates back-pressure if that is not the case, so that the queue at the receiver is capped.

Existing systems at any scale, single-node or distributed, implement some form of flow control. Token bucket algorithms are one of the common ways of implementing flow control in any kind of communication in fields such as wide-area networks [338], networks-on-chip [168], storage [332, 368, 382, 196], and network congestion control [59]. A typical example of a flow control mechanism, is TCP flow control based on the sliding window. TCP, as a connection-oriented transport protocol, implements flow control per connection. The flow control mechanism depends on the size of the available receive socket buffer communicated between the two endpoints with every packet exchanged. The sender uses this information in conjunction

with the current congestion window size to decide how many bytes to send, and sends the maximum amount allowed. Another more generic form of flow control is the way existing operating systems limit the number of processes or connections in the system so that the machine does not run out of available memory.

Flow control is a mechanism that depends on and controls the available memory on the receiving end, but also affects latency, since by limiting the available memory it also limits the size of the queue that can build up. As the offered load approaches the system saturation, latency goes to infinity since queuing time increases. A scheduling mechanism can improve latency and help achieve more throughput under the latency SLO, but it is not able to control how high latency can go. Flow control, on the other hand, is there to prevent system collapse and controls the maximum queue depth. So, it can cap the maximum latency at saturation.

However, existing flow control mechanisms are SLO-agnostic and operate at a very coarse granularity as a safety net. In this thesis we design and build SLO-aware flow control mechanisms and show how they can provide tail-tolerance guarantees.

2.4 Replication

The scale-out sharded pattern where each node serves a different part of the dataset and operates independently has certain limitations in terms of scalability and fault-tolerance. One replica might not be enough to serve all incoming load, while losing this replica can lead to data loss. Replication is widely deployed in the datacenter to tackle the scalability and fault-tolerance problems.

2.4.1 Replication for Fault-Tolerance

There is a wide-spread need for fault-tolerant datacenter systems. Systems such as Chubby [48], Zookeeper [157], and `etcd` [105] manage the hard, centralized state at the core of large-scale distributed services. For example, Kubernetes [57] and RamCloud [281] use etcd/Raft [276] for state management; Azure Storage [51], Borg and Omega [47], Ceph [362], and GFS [128] all have a built-in Paxos [174, 274, 207, 208, 209, 210] implementation to manage storage metadata and cluster state.

At the core of all fault-tolerant systems is a variant of a consensus algorithm that is used to implement distributed state-machine replication(SMR) [314]. State machine replication guarantees linearizability, namely all replicas receive and apply the updates to the state machine in the same order. Consequently, all replicas in the same group behave as a single machine. This field has been the subject of extensive academic and industrial research both from a theoretical [314, 207, 117] and systems [209, 48, 157, 276] point of view.

Most consensus algorithms can be split into two phases: the *leader-election* phase, where the participating nodes vote to elect a leader node; and the *normal operation*. Once a leader

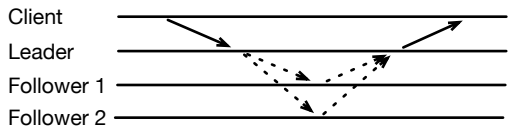


Figure 2.1 – Basic communication pattern to execute a client request in a fault-tolerant group of 3 servers. Solid arrows refer to messages from and to the client based on the service-specific API, while dotted arrows refer to SMR messages.

is elected, the leader is in charge of selecting one of the client requests to be executed and notifies the rest of the nodes about its choice. Once the majority of nodes are aware of this choice, the leader can commit and announce the committed decision, and the request can be executed.

Although the terminology (*e.g.,* acceptors and learners in Paxos vs. followers in Raft) and the specific semantics change across different implementations, the key point is that consensus algorithms generally operate in a 2-roundtrip communication scheme. In the first round-trip the leader announces the intention to execute an operation, and in the second it announces that the operation is committed, given the necessary majority.

In this thesis we will focus on Raft [276] and use Raft terminology. Raft is a consensus algorithm that depends on a strong leader and exposes the abstraction of a replicated log. The leader receives client requests, puts them in its log, thus guaranteeing a total order, and replicates those to the follower through an `append_entries` request. The followers append them to their logs and notify the leader. Then, the leader can execute the operation and reply to the client. The above interaction is summarized in Figure 2.1. The leader notifies the followers for its committed log index in the next communication round. The choice of Raft is crucial for HovercRaft's design (Section 12.3).

2.4.2 Replication for Scalability

Data replication, either in the form of caching or load balancing between replicated servers, is used to improve the latency and throughput of data accesses. Replication, though, improves scalability by trading off either availability or consistency, as the CAP theorem suggests [42, 81]. In such replicated systems, concurrent accesses and component failures lead to anomalies that must be handled either at the application layer or by the end-user herself [81].

Replication is therefore the fundamental mechanism used to offer either scalability or fault-tolerance, but not both at the same time. Replication for scalability requires compromises in the consistency model and can at best offer high-availability [24].

2.5 Internode Load Balancing

Having several replicas serving the same data introduces the problem of spreading out incoming load across those replicas. Load balancers encapsulate a set of servers behind a single virtual IP address and improve the availability and capacity of applications. Load balancing decisions, however, can severely affect throughput and tail-latency; thus, a significant amount of infrastructure is dedicated to load balancing [251, 96]. Load balancers can be implemented either in software [266, 96, 275] or in hardware [4, 108, 65, 251] and fall into two broad categories: layer-4 and layer-7 load balancers.

Layer-4 ("network") load balancers that use the 5-tuple information of the TCP or UDP flow to select a destination server. The assignment is done at a flow-level granularity and it can be static for stateful L4 load balancers or dynamic based on certain policies when the load balancer can maintain per connection state. Section 7.2 makes a deep dive to L4 load balancing to motivate our CRAB load balancing scheme.

Layer-7 ("application") load balancers come in the form of HTTP reverse proxies as well as protocol-specific routers implemented in software middleboxes [271] or SDN switches [62, 41]. These load balancers terminate the client TCP connections, use dynamic policies to select a target per request, and reissue the request to the server on a different connection. Layer-7 load balancers support many policies to decide the eventual RPC target, including random, power-of-two [253], round-robin, Join-Shortest-Queue (JSQ), and Join-Idle-Queue (JIQ) [232]. Layer-7 load balancers are ubiquitous at the web tier and can theoretically mitigate tail-latency better, due to their dynamic policies. However, they are less commonly deployed within tiers of applications to support µs-scale RPCs. The reasons for this are (i) the increased latency due to the extra hop (ii) the scalability issues introduced when all requests and responses flow through a proxy. Section 10.2 performs a queuing theoretic analysis on the different request-level load balancing policies, while Section 10.3 introduces *JBSQ* that deals with the above challenges.

Part I

Understanding and Measuring Latency Accurately

3 Thesis Methodology

Before getting into the system contributions, we describe the methodology used throughout this thesis to identify research problems, compare design solutions, and evaluate the existing and proposed systems. Our methodology is not specific to µs-scale datacenter systems and can be reused whenever the research problem is related to scheduling, system overhead elimination, and CPU and IO bottleneck identification. After initially describing our methodology in Zygos [299], elements of this methodology have been used by other researchers [178, 74].

3.1 System Modelling

Queuing theory has the beauty of abstracting away all the deployment and system-specific details and describes any scheduling problem as a mix of queues and workers. Queues implement different abstract policies, *e.g., Fist-Come-First-Serve* or *Shortest-Job-First*, while workers can run each individual task to completion or use processor sharing. Requests arrive based on a specific inter-arrival distribution and task service time follows another random distribution. We use Kendall's notation to describe the models, where in the following expression `A/S/n/K`, `A` is the inter-arrival distribution, `S` is the service time distribution, `n` is the number of workers and `K` is the policy implemented, *i.e.,* first-come-first-serve (FCFS) or processor sharing (PS). Thus, based on this notation a system can be fully described.

Then, through closed forms or discrete event simulation we can estimate metrics such as the average or tail latency, the number of elements in the queue, the time spend in each queue, *etc.*, independent of how complex the initial system is. So, modelling through queueing theory can help identify design flaws in existing solutions, compare different designs, and estimate the potential benefit for a proposed system before actually implementing it.

The process of modelling an existing system through queueing theory gets rid of system and communication overheads and the analysis operates on an idealized view of the system. For example, scheduling a new process in an operating system involves a timer interrupt, the scheduler logic that will decide the next process to run, the scheduler bookkeeping, and the

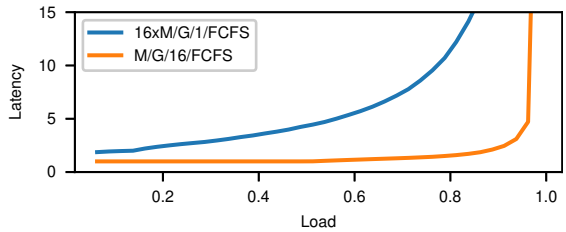


Figure 3.1 – Comparison between a 16xM/D/1/FCFS and an M/D/16/FCFS queueing model.

context switch logic. Depending on the implementation these can take a few µs. A scheduling decision in a queuing model has zero system overhead. Similarly, the communication between two end-points includes the transmission and propagation delays of the messages, unlike a model where messages can be exchanged without communication delays. So, the results of the queueing analysis can serve as an upper bound of the performance of the analyzed system, since they are overhead free.

Figure 3.1 plots the 99-th percentile latency for two basic queueing models. We consider a system with 16 workers that process one request at a time to completion. A multicore system 16 cores, a distributed system with 16 servers, or a super market with 16 cashiers would have exactly the same model. Requests arrive with as a Poisson process and have a fixed service time. There are two ways to split requests across workers. In the one case each worker has its own queue and incoming requests are randomly assigned to workers (`16xM/D/1`). Alternatively, there is only one queue where all requests wait for a worker to be available (`M/D/16`). We observe that single queue systems have better tail latency behaviour. Thus, theoretically it is better to design systems that depend on a single queue.

There is one main caveat in the use of modelling through queueing theory in the design of new systems. Despite being able to identify a superior model or scheduling policy, this model might incur severe system and communication overheads when implemented that mask its design benefits. Instead, a theoretically inferior model can perform better due to a simpler implementation. Identifying a model that performs better and does not suffer from implementation overheads can be challenging and requires considering several trade-offs. The performance difference between a model and the actual system implementation is crucial for µs-scale computing where system inefficiencies become apparent.

In this thesis, we used modelling both to understand and evaluate the existing state of the art, and to propose new scheduling policies. In Section 6.2 we model how servers can serve incoming requests and in Section 6.3 we see how those models map to state of the art systems. This analysis led to the design of ZYGOS (Section 6.4). In Section 10.3 we model how requests can be load balanced across different replicas using existing load balancing techniques and identify the benefits of our proposed *Join-Bounded-Shortest-Queue* policy through simulation before implementing the system. In Subsection 7.2.2 showcase the impact of distributed versus centralized load balancing decisions on tail latency that motivated the

design of CRAB (Section 7.3).

3.2 Synthetic Microbenchmarks with Known Boundaries

To evaluate the existing or newly proposed systems we extensively used synthetic microbenchmarks, except for real applications. In a synthetic microbenchmark the server does not perform an actual task, but instead spins for a certain amount of time before returning a reply. Using this approach we can independently control and parametrize the request size sent by the clients, the request service time, and the reply size sent by the server. The values for these parameters come from idealized probabilistic distributions described in closed forms, *e.g.,* a Poisson process in which the gaps between requests follow an exponential distribution.

Using synthetic micro-benchmarks improves experiment reproducibility, while it helps studying each system bottlenecks independently. When using a real application for system evaluation, the benchmark results highly depend on how the application is configured. Configuring datacenter applications according to production workloads can be challenging [14], while resource isolation is a research topic on its own [231]. Also, interference for shared resources , *e.g.,* memory interference, while not relevant for the specific benchmark, can affect the final results. A synthetic microbenchmark, on the other hand, is only described by three independent distributions: the request size, the service time, and the reply size, while it eliminates interference at the application layer, since it only uses a busy loop on the CPU. This enables us to focus on independent bottlenecks and evaluate them separately while knowing what the upper bound for each benchmark is. Thus, we can identify how far from ideal the system is. For example, for a CPU-bound application we configure a microbenchmark with small requests and replies that have a fixed size, *e.g.,* 8 bytes, and only control the service time. With an average service time of 10µs on a single core system, we know that the maximum achieved throughput is 100k requests/sec. Failing to reach this throughput can only be attributed to system overheads that need to be fixed or at least justified. Similarly, for an IO-bottlenecked application, we configure small requests and service times, but large replies. For example, having an average reply size of 6kB on a server connected with 10G NIC can only scale to 200 kRPS.

Despite that the use of synthetic service-time microbenchmarks is not novel and already widely used, *e.g.,* [220, 232], our proposed methodology uses them in combination with their equivalent queueing models. Synthetic microbenchmarks and queueing models use the same level of abstraction: well-known inter-arrival and service time distributions. Thus, we can simulate a queueing model and then run the equivalent synthetic microbenchamrk on the actual system and compare the results. Plotting the model and the microbenchmark results on the same latency versus throughput graph reveals two things: i) how well or bad does the model track the system; ii) how much worse does the actual system perform compared to the model due to unmodelled overheads. In a proper modelling analysis the model and the system should have the same curve shape. The vertical and horizontal difference between the

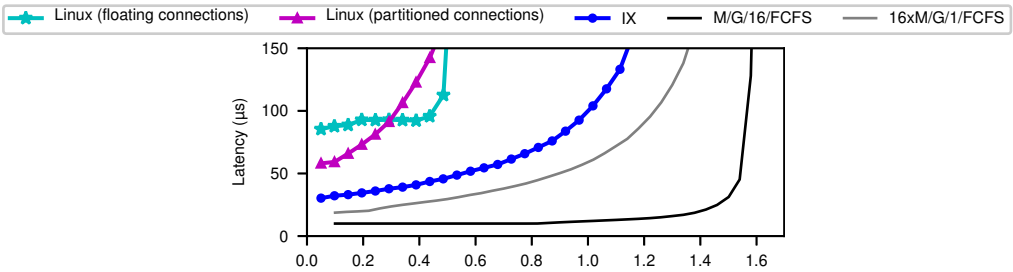


Figure 3.2 – Combining and comparing queueing models with synthetic microbechmarks

two curves exactly quantifies the cost of the system overheads.

We showcase the above analysis with an experiment. We run a CPU-bound synthetic microbenchmark with a fixed service time of 10 µs and Poisson arrivals, on Linux and on IX [33]. On Linux, we use two modes, one in which each connection is associated with a single core, and one where any core can server any connection. The first mode corresponds to an `16xM/D/1` model, while the second corresponds to a `M/D/16` one. IX is equivalent to an `16xM/G/1` model due to static partitioning of resources.

Figure 3.2 plots the experiment results, as well as the two model curves from the previous figure. We make three main observations that showcase the benefits of the methodology

- The models captures the equivalent system behaviour as it can be seen by the shape of the curves. The `M/G/16` curve has the same shape with the Linux (floating connections) system configuration, while the `16xM/G/1` model tracks well the Linux (partitioned connections) and IX as expected.
- There is a performance difference between Linux (partitioned connections) and IX despite following the same model. This performance difference is quantified by the distance between the system curves and the model curve. IX that is more efficient and has less system overheads performs closed to the model.
- The figure showcases the aforementioned caveat. Linux (floating connections) despite following a superior model performs worse than IX due to system overheads, proving the point that better design does not always perform better on the actual system. Note that if we increase the service time the impact of system overheads diminishes and the system and model curves converge.

4 Lancet: A self-correcting Latency Measuring Tool

4.1 Introduction

Today's webscale datacenter applications such as search, social networking, and e-commerce all rely extensively on the decomposition of online, data-intensive queries into smaller sub-queries that process data directly from the memory of hundreds or thousands of tightly-interconnected servers to ensure service-level objectives, scalability and availability [223, 80, 81, 14, 219]. The combined advancements in hardware technology (*e.g.,* 10-100Gbps NICs, cut-through switches, NVMe), system software (*e.g.,* dataplanes [290, 34]), and data management systems (*e.g.,* in-memory databases and key-value stores [337, 281, 90, 271]) now allow µs-scale interactions between application components [31]. The increased number of components involved in a single query and the extensive use of high fan-in, high fan-out patterns have shifted the performance focus to tail-latency considerations [80].

This emerging µs-scale computing era is characterized by new key performance metrics such as the tail-latency service-level objective (SLO), *e.g.,* $99^{th} percentile \leq 500\mu s$ [188, 34]. To put this into perspective, 500µs is one order of magnitude longer than an in-memory relational database processing TPC-C [337] and two-to-three orders of magnitude longer than basic operations on a key-value store [90, 229, 281]. Yet, it is shorter than an operating system quantum, a TCP retransmission timeout, or the DVFS governor's reaction time [230]. This requires complete rethinking of traditional assumptions about systems, stacks, protocols, and applications [31, 309, 281, 90]. A large body of research focuses on the systematic characterization and reduction of tail latency effects [188, 230, 85, 84, 223, 182, 187, 155, 34, 299].

While throughput can easily be measured, tail latency is harder to capture and characterize in a statistically meaningful manner, as it depends on a number of factors beyond the workload itself. These factors include the choice of a tool with overheads and biases, its precise configuration, and the experimental methodology. The literature describes many pitfalls specific to latency, *e.g.,* Treadmill [379] discusses situations in which: (i) the inter-arrival request distribution does not match the production environment; (ii) the measuring methodology silently masks some tail behaviors; (iii) the measuring tool affects the measured end-to-end latency

because the measuring granularity is too coarse-grained or because the clients are overloaded.

This matches our own experience in building and evaluating multiple research systems for µs-scale computing [34, 299, 300, 202], which used either modified versions of Mutilate [219] or home-grown latency-measurement and load-generation tools. While these tools measured the tail latency of our systems as a function of the load, we were required to unscientifically tweak a large number of workload and system parameters in an ad-hoc manner by: (i) repeatedly increasing the number of load-generating clients until stability; (ii) repeatedly increasing the number of outstanding requests (*e.g.,* number of connections) until the tail-latency diverges at saturation, as expected in an open-loop process; (iii) and last, but not least, running each experiment "longer" with the hope of reducing result jitter.

This chapter introduces LANCET, a self-correcting latency measurement tool designed to measure, in a statistically sound manner, the end-to-end tail latency of remote procedure calls in a testing environment. LANCET is self-correcting as it relies on on-line statistical tests to determine situations in which tail latency cannot be accurately measured. This includes situations when (i) the workload configuration, and in particular the number of client connections, leads to closed-loop behavior; (ii) the infrastructure (*e.g.,* number of machines) cannot generate the desired load without introducing client bias; (iii) the service time of the workload itself is heavy-tailed distributed.

Because it relies on statistical methods within its control system, LANCET is also easier to use than existing tools. While the scientist specifies the infrastructure used for an experiment (*e.g.,* number of client machines), and the workload itself (*e.g.,* mix reads and writes, distribution of keys and values, number of client connections, maximum number of outstanding requests per connection *etc.*), LANCET then automatically determines, using statistical tests, what can be measured and at which confidence interval. LANCET's control system internally sets additional experimental parameters such as the duration of the experiment and its warmup phase.

Finally, LANCET relies on state-of-the-art, hardware-based measurement techniques that combine NIC timestamping in hardware and userlevel matching of packets to RPCs. This approach noticeably eliminates the client bias, and increases the accuracy of individual measurements without creating a long-term dependency on immature kernel-bypass protocols stacks and libraries.

This chapter describes the methodology, design, and implementation of LANCET, with the following novel features:

- LANCET measures the open-loop tail latency of a workload using only two user-provided parameters: the target load level and the desired confidence interval at the target tail percentile. For this, it relies on proven statistical methods such as hypothesis testing to configure the experiment methodology parameters.
- LANCET is self-correcting and reports "N/A" when no statistically-sound tail latency

can be measured. This can be due to limitations in workload specification, client infrastructure, or because the service time distribution has high variability.

- LANCET clearly separates (i) the methodological considerations, implemented by the LANCET controller, (ii) the measurement tool, implemented by a combination of agents, and (iii) the workloads and application-level protocol support, implemented in an extensible manner by the LANCET agent's internal API.

- LANCET is designed with stability and production deployment in mind, with a focus on Ethernet-based protocols. It therefore uses exclusively the standard Linux kernel-based implementations of networking protocols. For applicable NICs, LANCET supports hardware-based timestamping to measure TCP-based RPC latencies for improved measurement accuracy. Our work demonstrates that kernel-bypass is not necessary to achieve precise µs-scale client-side measurements.

Our evaluation of LANCET with workloads with synthetic service times demonstrates that it (i) automatically identifies the right number of samples necessary for the target experiment accuracy and result convergence; (ii) accurately reports the latency distributions for workloads with service time as short as $\bar{S} = 1\mu s$; and (iii) provides substantially more accurate results than Treadmill [379] and Mutilate [219], state-of-the-art tail latency measurement tools.

LANCET is open-source and can be found at `https://github.com/epfl-dcsl/lancet-tool`.

4.2 Background

The accurate measurement of the latency of any software application serving RPCs requires the appropriate combination of metrics, tools, workloads and experimental methodology: (i) the metrics determine which type of latency is being measured for a certain load, whether mean, median, tail (*e.g.*, 99th percentile), or the empirical cumulative distribution function (ECDF); (ii) the choice of tool determines the precision of the benchmark; (iii) the choice of workload determines the degree of realism, generality, and relevance of the experiment; (iv) the choice of methodology determines the overall soundness of the results, their accuracy, and reproducibility.

The high-level process is straightforward: the tool acts as an RPC client which generates requests for the system(s) under test. The tool timestamps requests and corresponding replies to determine the end-to-end latency. The requests themselves are determined in a workload-specific manner (*e.g.*, a mix of `get` and `set` with a specific distribution of keys). The individual request inter-arrival time typically follows a Poisson process for a given rate λ. For a fixed rate λ, scientists typically report the full ECDF or the complementary cumulative distribution or tail distribution (CCDF), often on a log scale, to highlight the tail latency levels (99th, 999th, *etc.*).

To study the impact of load on latency, scientists repeat the fixed-rate experiment for different λ and report the tail latency as a function of the load [219, 299, 34, 85, 188, 231, 138, 379, 188]. Finally, for dynamic experiments that mimic daily datacenter patterns, the tool dynamically adjusts λ according to a diurnal (or accelerated) time pattern [34, 300, 14, 340].

4.2.1 Taxonomy of tools

We attempt to make a taxonomy of the existing tools for generating load and measuring latency, the techniques used and the main design decision.

Packet vs. RPC generators: At the highest level, latency measuring tools can be easily classified into packet generators, which measure a network device or a network function, and application RPC generators, which measure a server.

Packet generators use stateless network packets to measure the throughput and the latency of datacenter network equipment such as switches and routers as defined in RFCs [236, 40]. These tools can be implemented to achieve different levels of precision in software. For example, MoonGen [98], TRex [64], and netperf [264], rely on hardware timestamping facilities in modern NICs (*e.g.,* MoonGen) or use custom hardware appliances such as Spirent [323] or IXIA [163].

Application RPC generators measure the latency of client-server interactions using protocols such as `http` or memcached's binary protocol, typically implemented on top of TCP or RDMA connections. These tools provide advanced workload-generation capabilities. For example, Mutilate [219, 261] can model Facebook's various uses of key-value stores [14], YCSB [70, 372] can generate a Zipfian distribution of keys, and CloudSuite [114, 68] offers a mix of applications. For the rest of the chapter, we will focus on RPC generators.

Open-loop vs. Closed loop: There are two main ways to control the flow of requests to the target. An open-loop system models $n = \infty$ clients that send requests to the target according to a rate λ and an inter-arrival distribution, *e.g.,* Poisson process. A closed-loop system bounds the maximum number of possible outstanding requests at any given time. The distinction between an open and a closed loop system is a property of a specific deployment and the same system can be deployed under different scenarios, *e.g.,* a key-value store may serve only a few blocking clients (*i.e.,* closed-loop) or thousands of application servers, which is best modelled as open-loop. Testing for the right scenario is crucial because open-loop systems can lead to large queuing, and thus longer tail latencies, whereas closed-loop tail latencies are typically bounded by the number of possible outstanding requests. Tools such as Treadmill and Mutilate are open-loop systems, while others such as YCSB are closed-loop systems.

Generating the necessary load: Precise tools are typically used to evaluate the benefit of innovations in new hardware, protocol designs, kernel bypass architectures, networking

stacks, operating system configurations, or applications. Leading research systems today can deliver high-throughput solutions that easily scale to millions of requests per second, even on commodity hardware. As a consequence, the load-generation and latency-measuring tools, which typically run on reference vanilla Linux infrastructure, must be distributed on multiple client machines to saturate a single server [219, 34, 299].

Multi-machine setups follow two basic design patterns. First, in symmetric generators, all client machines generate load and measure latency. Then, an external agent accumulates and processes the collected results to report the aggregated verdict. This category includes YCSB [70], Treadmill [379], CloudSuite [114], memaslap [249], *etc.* Unfortunately the open-source versions of these tools provide no coordinator or aggregator script to run them in a distributed fashion.

Second, the asymmetric design splits the client machines between load-generating and latency-measuring. The bulk of the load is generated by client machines that generate requests according to a specific inter-arrival distribution, *e.g.,* Poisson arrival, in an open-loop manner without measuring latency, while a separate, dedicated client machine makes closed-loop requests to the same server and measures its latency. By reducing the system load on the latency-generating thread, such tools reduce client bias in the measurement. Mutilate [219] is the most well-known tool in that category.

Point of measurement: Latency can be measured at different points in the system resulting in different levels of accuracy. This includes the actual wire, the NIC, the Ethernet driver, the in-kernel socket layer, or the application itself. The point of measurement has a large impact on precision. According to Primorac *et al.* [301], (i) the packet generators using hardware-based NIC timestamping such as MoonGen can accurately measure the latency of stateless network functions up to the 99.99^{th} percentile whereas (ii) the best software solution relying on kernel bypass can only measure up to the 99th percentile, and (iii) the solutions relying on the traditional networking stack should not be used at all for µs-scale latency measurements.

NIC-based timestamping is available on mainstream NICs. Intel NICs, such as 10Gbe 82599 and x54, or 40Gbe x710, implement hardware timestamping only to support IEEE 1588 Precision Time Protocol [95]. This restricts the type and amount of packets that can be hardware-timestamped. The MoonGen packet generator takes advantage of this precise, yet restrictive mechanism. The Mellanox NICs, *e.g.,* ConnectX-4 [247] or newer, offer general-purpose hardware timestamping support to all incoming and outgoing packets. The Linux kernel provides support for hardware timestamping via the Linux socket interface, yet deriving RPC timestamps from packet timestamps is challenging, as later described.

Another way to increase precision and reduce jitter is to leverage kernel bypass and NIC polling at the client. Tools such as MoonGen and T-Rex use the DPDK [88] toolkit for better performance and precision. Unfortunately, kernel bypass limits application and protocol support, and requires using less-proven protocol stacks as part of the experiment.

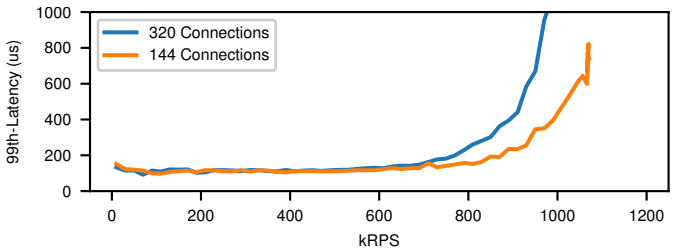


Figure 4.1 – 99th percentile latency for memcached USR measure with Mutilate with 144 and 320 connections with 1 outstanding request per connection.

Reporting results: From a methodology perspective, most tools depend on histograms to compute latency percentiles, thus avoiding keeping all the recorded latency samples. Histograms with fixed bucket sizes, as used by Mutilate, can affect the reported results by masking tail phenomena, if not configured properly. Some tools such as Treadmill [379] propose a user-defined calibration phase to determine the bucket allocation. Other tools, such as TailBench [188], use dynamic histograms, whose bucket sizes change over the execution of the experiment. Finally, few tools, *e.g.,* Mutilate, allow collecting all latency samples and save them in a file to be used for plotting the ECDF.

4.2.2 Configuration burden

Load generators put the methodological burden on the scientist who configures it. For example, a scientist using Mutilate must first determine the time for each load experiments (default=5s), which must be long enough to be statistically sound; then specify the number of machines, threads and overall number connections for the load-generating agents, and the maximum number of outstanding requests per connection; and finally specify the configuration of the latency-measurement agent, which operates as a closed-loop with one outstanding request a time.

This configuration setup has subtle implications as (i) increasing the number of machines reduces client bias [379]; (ii) increasing the number of open sockets reduces the throughput of the server because of operating system overheads [34]; (iii) increasing the maximum number of outstanding requests per socket allows for batching and increases throughput; (iv) the product of the number of connections × the number of outstanding requests must be larger than the bandwidth-delay product of the workload if the scientist wishes to measure the open-loop tail latency of the service.

Figure 4.1 illustrates the challenge via the study of an out-of-the-box memcached/Linux deployment with Mutilate configured with 320 and 144 connections with one outstanding request each. We report the 99th tail latency as a function of the load. The orange curve (144 Connections) operates as a closed-loop, with the clients unable to generate the target rate, and without ever saturating the server. The lower reported tail latency is merely a reflection of

Latency Experiment Concerns			
Workload		**Methodology**	**Measuring Tool**
transport protocol	connection balance	system stability	workload-compliant
application protocol	open/closed queueing	unbiased result processing	methodology-compliant
request types and ratio	outstanding reqs/conns	result convergence	measuring bias free
connection count	inter-arrival distribution	distribution coverage	

Table 4.1 – Classification of concerns related to running a latency experiment into workload, methodology, and measuring tool-specific components. We advocate that the Workload column has to be user defined, while Methodology and Measuring Tool columns have to be handled systematically by the measuring framework.

the limited number of outstanding requests. This experiment can lead to false conclusions, *e.g.,* on the maximum throughput that meets a SLO (*e.g.,* $\leq 300\mu s$).

4.2.3 Statistics Background

This section provides the sufficient background to understand our use of statistical methods in LANCET.

Hypothesis testing: Statistical testing follows a specific thought process. Initially, the statistician formulates a null hypothesis implying that there is no relation between two populations and the observations are the results of pure chance. She then identifies a test statistic that can assess the truth of the null hypothesis and computes the p-value. p-value gives the probability of the given test statistic resulting in the observed value if the null hypothesis is true. The smaller the p-value, the stronger the evidence against the null hypothesis. Finally, she compares the p-value to the α value, which corresponds to the level of confidence. If the p-value is less than α, she rejects the hypothesis and therefore conclude that the effect she observed was not due to random chance.

LANCET uses the following tests. First, the **Anderson-Darling** test checks whether a group of samples comes from a certain probability distribution and was chosen because it is less sensitive to outliers compared to similar tests, *e.g.,* the Kolmogorov-Smirnov test [214]. We use that to validate the inter-arrival request distribution. Second, the **Augmented Dickey Fuller** test [86] checks a series of samples for stationarity. We use the ADF test to determine the duration of the warm-up phase and whether the experiment results change over time. Finally, we use the **Spearman** rank correlation coefficient and the associated p-value to check if a series of samples is autocorrelated when checking for *iid*-data.

IID-data: Most types of hypothesis testing or general statistical processing, such as the calculation of confidence intervals, require samples that are *independent and identically distributed (iid)*. When running a latency experiment, latency samples are naturally identically distributed since they come from the same target server. Sample independence, though, is challenging to

meet because of queuing effects. The end-to-end latencies of two requests that are queued back-to-back are dependent because the latency of the latter request includes the service time of the prior. While independence cannot be taken for granted, it can be tested, with autocorrelation being the standard way to check independence for a series of samples.

Confidence Intervals: We focus here on the confidence intervals for tail latency of a *single* execution, assuming that the system environment remains identical and stable during the entire experiment. The confidence intervals for a distribution's percentiles can be computed in closed form when the data are *iid*. The formula identifies, with a certain level of confidence, two threshold values that belong to the collected samples, between which the value for the specific percentile is expected to be found. Formulas 4.1, 4.2 give the indices of those two threshold values in the sorted of collection of samples [214] for a certain confidence level γ.

$$j \approx \lfloor np - \eta\sqrt{np(1-p)} \rfloor \tag{4.1}$$

$$k \approx \lceil np + \eta\sqrt{np(1-p)} \rceil + 1 \tag{4.2}$$

where n is the number of samples, p is the percentile, and η is defined as $N_{0,1} = \frac{1+\gamma}{2}$. For example, for 10,000 *iid* samples ($n = 10000$), the confidence interval for the 99-th percentile with 95% confidence ($\gamma = 0.95$, so $\eta = 1.96$) will be between the values with indices $j = 9880$ and $k = 9921$.

Note that determining confidence across different executions of the same experiment is challenging as the system's boot-time and application initialization can have a persistent effect on performance, leading to the hysteresis problem described in Treadmill [379].

4.3 Experiment Decomposition

Our goal is to build a latency-measuring tool that is precise and simplifies the configuration burden discussed in Subsection 4.2.2, with the explicit objective to identify situations in which the configuration cannot lead to a statistically meaningful result.

Table 4.1 classifies concerns related to a latency experiment into three main categories: workload, methodology, and measuring tool. These concerns often correspond to user-defined parameters in most of the existing tools and can be easily misconfigured. This decomposition will guide the design of modular, self-correcting latency-measuring tools. We claim that the workload-specific parameters have to be user defined, otherwise the experiment is insufficiently described. The methodology and measuring tool concerns have to be systematically managed by the measuring framework to reduce the pitfalls induced by the user misconfiguration.

Workload: The first aspect of a latency experiment is the actual workload and a large set of the configuration parameters refer to the workload specification. The experiment workload is both application- and deployment-specific, meaning that the same application should be tested differently if the deployment environment is also different. The workload includes the application specific parameters (*e.g.,* `get:set` ratio, request size distributions, TPC-C request mix, *etc.*), the application-level protocol (*e.g.,* `HTTP`, binary memcached, *etc.*) and the network-level protocol (*e.g.,* UDP vs. TCP). The definition of the workload also includes the client assumptions, *i.e.,* the number of expected client connections, the maximum number of outstanding requests per connection, and whether clients operate in an open- or a closed-loop system.

Critically, the specification of the workload is independent of the measuring tool , but affects the results, which could lead to unrealistic or wrong conclusions. For example, one cannot meaningfully report the open-loop tail latency of a workload with an insufficient number of connections, or insufficient outstanding requests per connection.

Measuring methodology: The second aspect of a latency experiment is the methodology, which describes how the latency samples are collected and processed. Examples of configuration parameters that are relevant to the methodology are the experiment duration, the number of collected samples, and the number and size of the histogram buckets. Reducing the number of configuration parameters related to methodology is a major goal of our design.

Regarding the latency sample collection, a good methodology should first ensure that the system under test is in a steady state to avoid measuring transient phenomena. Then it should ensure that the collected results converge and that all desired tail behaviors are covered. Finally, during the result processing, it should avoid adding statistical bias, *e.g.,* by the misconfiguration of histograms.

Measuring tool: Finally, the last part of a latency experiment is the actual client software used to collect the latency samples. Examples of parameters related to the tool are the number of client machines or threads used in the experiment, and whether hardware or software timestamping is used. The tool should be able to implement the specific methodology, generate the target workload accurately, and measure latency without adding too much client bias.

4.4 Design

4.4.1 LANCET infrastructure

Figure 4.2 shows the basic LANCET overview, which splits the methodology from the actual measuring tool and workload generator according to Section 4.3. LANCET is a by-design distributed tool that consists of a coordinator (C) and various measuring agents. The coordinator is in charge of the experiment methodology (see Subsection 4.4.3) and communicates with

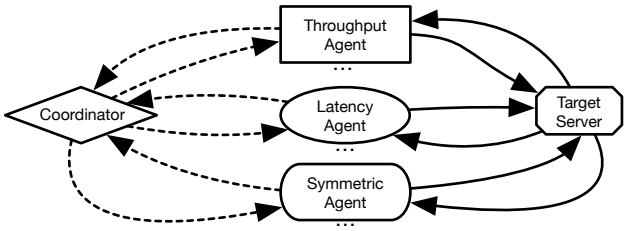


Figure 4.2 – Lancet's architecture depicting a coordinator (C), throughput agents (TA), latency agents (LA), symmetric agents (SA), and the target server under test. The dashed arrows correspond to the LANCET API while the solid ones are application RPCs

Table 4.2 – The LANCET coordinator API with the information returned by the agents on each call. For the `get_throughput` and `get_latency` requests, the agents also reply information related to the Inter-Arrival distribution (IA), the latency Confidence Intervals (CI), and whether the collected samples are stationary and *iid*. If they are not *iid*, the reply contains the target sampling rate necessary to get *iid* data.

Request Type	**Request Params**	**Reply**
start_load	load (rps)	ACK
start_measure	#samples sampling rate(sr)	ACK
get_throughput	None	Throughput (rps) Correct IA (T/F)
get_latency	None	Latency CI Stationary (T/F) IID (T/F, sr)
exit	None	ACK

the agents over the LANCET API (Table 4.2). The measuring agents drive the workload via application RPCs generated based on application-specific random distributions. The agents also measure latency precisely, identify cases of workload violations, and run statistical tests.

Figure 4.3 describes a typical agent state transitions triggered by the coordinator via the API for a fixed-load experiment. From an idle state (*Idle*), the agent transitions into the loading phase (*Load*), where it attempts to issue l requests per second to the server. During that period the agent does not record latency. The agent eventually transitions into the measurement phase (*Measure*) specified by a sampling rate (sr) and a number of latency samples to collect (s). The agent can stay in that state while the sr and s parameters can change. Finally, the coordinator decides to terminate the experiment (*Terminate*) via an `exit` message. At any point in time, while the agent is in the *Load* or *Measure* phase the coordinator can ask for the current throughput and latency.

4.4.2 Measurement options

Figure 4.2 shows that LANCET implements three agent types, selected to match the capabilities of the available hardware, the measuring methodology and the target experiment granularity. This way LANCET can support both symmetrical and asymmetrical deployments described in

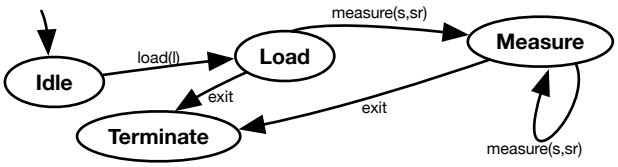


Figure 4.3 – Lancet agent's state transition. Arrows represent messages from the coordinator.

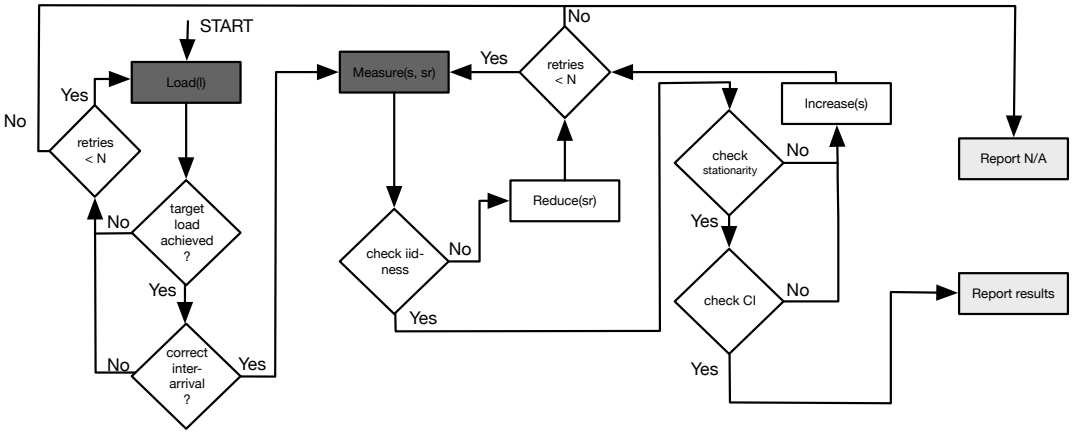


Figure 4.4 – LANCET's experiment methodology implemented by the coordinator. Dark grey boxes correspond to messages from the coordinator to the agents. Light grey boxes show the experiment end.

§ 4.2.1.

LANCET uses the asymmetrical model when the latencies are captured in software. This model reduces jitter by dedicating cores and even machines to only measure latency. The drawback is that the experiment collects fewer samples per time period. Furthermore, special care must be taken to ensure that the collected samples are representatives of the workload. For example, a latency agent should open multiple connections (*i.e.,* emulate multiple clients) to ensure that a server configured with an RSS NIC will use all cores.

LANCET uses the symmetrical model when the NIC offers the capability to timestamp all incoming and outgoing Ethernet frames and the Linux operating system exposes the information to userspace (v4.14+ kernels). LANCET associates the hardware timestamping of packets to the end-to-end latency of RPCs. This is not straightforward because of the inherent mismatch between the stream-oriented TCP protocol and the message-oriented RPCs. Implementation details follow in Section 4.5.

4.4.3 LANCET's self-correcting methodology

LANCET's primary contribution is its novel, self-correcting methodology which follows the experiment decomposition and split of concerns described in Table 4.1, and tries to systematically and based on statistics, identify: (i) when the server is in a stable state to start measuring

latency (managed by the user-defined warm-up time in other tools); (ii) if the collected latency samples converge and whether tail phenomena are fully covered (controlled by the user-defined experiment duration in other tools); (iii) how to process the collected samples and report latency without introducing statistical bias (histograms are mainly used for that purpose in other tools); (iv) the confidence intervals of the latency results (unlike most tools which simply report latency percentiles).

Figure 4.4 illustrates the state machine transitions of the coordinator when measuring the open-loop tail-latency of a server under a certain load. To run such an experiment, the scientist needs to provide, apart from the necessary workload specification (first column in Table 4.1), the following:

- the target load (`l`).
- the target confidence interval for a specific latency percentile, *e.g.,* 10µs interval for the 99th percentile with 95% confidence.

The output of such an experiment will be either the tail-latency percentiles with the corresponding confidence intervals or an indication that the specific experiment cannot be executed because some of the assumptions are violated, *e.g.,* the target load cannot be reached, the service time has high variability and the computed latency confidence interval is wider than the target, the client does not respect the workload specifications, *etc.*

System Stability: Initially, the methodology ensures that the target load can actually be reached before starting measuring latency, thus eliminating transient phenomena. Then, the methodology ensures that agents load the server while respecting the workload's specified inter-arrival distribution. This second confirmation is essential to avoid reporting misleading latencies. For this, every agent records the inter-transmission intervals of requests by recording request transmissions, ideally in hardware, but if necessary at the socket interface. Every agent runs an Anderson-Darling test to check whether the inter-transmission intervals follow the target inter-arrival distribution, *e.g.,* exponential in the case of Poisson inter-arrival. The controller exits the system stability step only when the load is reached according to the correct inter-arrival distribution.

Unbiased Result Processing: Each agent collects, according to the parameters (s, sr) set by the controller, s samples, each randomly sampled among the RPCs at rate of sr, *e.g.,* collecting 10,000 samples with a 1:20 sampling rate would require ~200K RPCs. Sampling is necessary because the collected samples need to be *iid* to compute the confidence interval correctly. Computing confidence intervals on non-*iid* data will underestimate their size.

The *iid*-ness is confirmed or rejected by computing the autocorrelation of the collected latency samples sorted by their transmission time. To do so, latency-measuring agents compute the Spearman correlation of the collected latency samples shifted over time. We leverage the

associated p-value to determine whether the correlation is significant or not. This correlation being significant implies that data that are close in time depend on each other, which is the result of them being queued back to back in the servers queue.

A way to reduce the autocorrelation is to decrease the sampling rate. The LANCET built-in parameters initialize the measuring phase with 10,000 collected samples with a first sampling rate of 1:5. If the autocorrelation is non-significant, the latency measuring agents report that the samples are *iid*. Otherwise, they report how much to reduce the sampling rate to achieve non-significant correlation. The latency measuring agents report the Spearman correlation co-efficient back to the coordinator as part of their latency results. To do so, the agents compute the autocorrelation for different lags and report the one that leads to a non-significant correlation. Based on the agents' replies, the coordinator decides whether to proceed to the next state or reduce the sampling rate accordingly if it fails to confirm *iid*-ness.

Result Stationarity: The methodology needs to identify whether the number of samples collected is sufficient for the results to converge to a stable distribution of latencies that does not change over time. To ensure stationarity, the methodology leverages an Augmented Dickey Fuller test [86]. Each latency-measuring agent sorts the collected latency results based on their transmission timestamp and runs the test. Again, the latency measuring agents report the result of the test to the coordinator. In cases where lack of stationarity is detected, the coordinator decides to increase the number of samples by 10,000 and retry. Otherwise it proceeds with the next check.

Determine the confidence interval: Finally, the methodology has to check if the results converge within the target confidence interval size. For that, we use the Formulas 4.1 and 4.2. Each latency measuring agent reports the confidence intervals for the latency percentiles to the coordinator. The coordinator ensures that the intervals from different agents are overlapping and computes their average. If the final confidence interval is wider than the user-selected target, the coordinator increases the number of samples by 10,000 and continues the experiment.

Termination: If the target confidence is reached, the coordinator finishes the experiment and reports the final latency percentiles with the equivalent confidence intervals. If the coordinator cannot reach the target confidence after a fixed number of failed retries, or if the experiment duration is above a certain threshold, it terminates the experiment, and reports that the specific experiment is not conclusive, the reasons why, along with the collected results so far.

4.5 Implementation

In addition to the design goals of Section 4.4, LANCET was implemented with robustness and long-term relevance in mind. LANCET is therefore built purely on functionality provided by the Linux kernel, using built-in drivers and protocol stacks. During development we identified

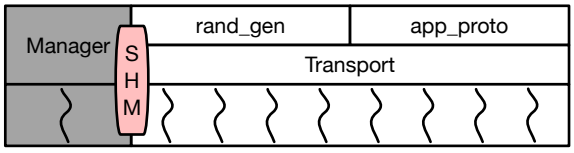


Figure 4.5 – The structure of a LANCET agent. The grey part corresponds to the Python control plane, while the white part corresponds to the C dataplane communicating over shared memory (SHM).

some inconsistencies regarding hardware timestamping in the Linux kernel; our patch was merged in Linux kernel 4.19.4 [235].

The LANCET coordinator (Figure 4.2) is in charge of deploying the agents, communicating with them over sockets, driving their state machine according to Figure 4.3, and implementing the methodology of Figure 4.4. The coordinator is implemented in Golang. It relies on *goroutines* for easy distributed coordination and failure management, and consists of ~1000 lines of code. From those lines, ~300 of them implement the methodology described in § 4.4.3 and the rest implemented the LANCET API to communicate with the agents, manage collected results, *etc.* Thus, implementing a new coordinator logic for different experiment methodologies is relatively easy.

We implemented three different agents that can be used according to the available hardware, the measuring methodology, and the necessary experiment granularity. Our agents can achieve better measuring granularity compared to previous tools and can be used in both a symmetrical and asymmetrical deployment, independently of the available hardware. The agents are implemented in a combination of C and Python, and can be easily extended with new transport and application protocols.

Figure 4.5 depicts the structure of a multi-threaded LANCET agent. Each agent is split between a Python control plane and a C data plane communicating over shared memory. The Python control plane is in charge of communicating with the coordinator and performing the statistical computations. The choice of Python allowed us to take advantage of the rich Python ecosystem using libraries such as NumPy and SciPy. The choice of C for dataplane gave us direct access to low level socket APIs and reduced the client overhead. LANCET lancet currently supports TCP, UDP, and R2P2 [202] as transports, and Memcached, Redis, and HTTP as application protocols.

Throughput Agent: This agent leverages `epoll_wait` to manage connections and is in charge of loading the server without measuring latencies. It is used only in asymmetrical deployments in cooperation with one of the two following agents that can measure RPC latency.

Latency SW-timestamping Agent: This agent depends on software timestamping and does not have any hardware dependencies. It improves the measuring precision over other software-

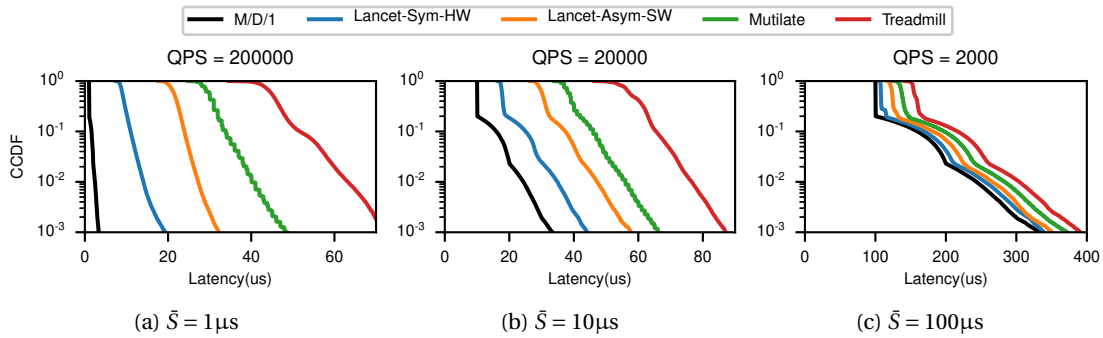
based tools, though, by leveraging the busy polling functionality introduced in Linux 3.11. Specifically, the `SO_BUSY_POLL` socket option allows blocking system calls to poll the NIC instead of depending on interrupts. While still dependent on userspace timestamping, this agent reduces client bias and measures latency with similar accuracy to kernel-bypass approaches. The blocking nature of this agent limits the load and the inter-arrival distribution of requests the agent can achieve. Consequently, this agent can only be used in asymmetric setups in conjunction with throughput agents that generate the necessary target system load according to the expected inter-arrival distribution.

Symmetric HW-timestamping Agent: Finally, we implemented a symmetric agent that leverages hardware timestamping to measure RPC end-to-end latency. This agent depends on the Linux kernel functionality for hardware timestamping added in kernel 4.14 for TCP. It also requires a NIC that timestamps all the incoming and outgoing packets, *e.g.*, the Mellanox ConnectX-4 [247]. The preferred deployment is based on symmetric HW-timestamping agents, as they improve on the latency agent in terms of precision and they can scale throughput while the coordinator can symmetrically collect results from all client machines, thus increasing the experiment accuracy.

The most challenging part of the implementation was the attribution of RPC latencies when requests and replies are layered on top of the stream-based TCP protocol, as used in the popular protocols, such as Memcached.

For TX timestamps, the Linux kernel provides an asynchronous API to collect timestamps, returning asynchronously one timestamp for each `sendmsg` system call. The notification is propagated to the userspace through an `EPOLLERR` for the equivalent socket that is handled by `epoll_wait`. Along with the timestamp, the kernel also returns the number of the last transmitted byte this timestamp corresponds to. For example, if the first request has a size of 20 bytes, the notification will mention that this timestamp is associated with byte 20. For the second request of the same size, the notification will mention byte 40, *etc.* The same information is maintained by LANCET in userspace for validation purposes, and to deal with cases of coalescing or resubmissions.

The kernel provides a synchronous API to retrieve the RX timestamp: the RX timestamp is part of the metadata to the `recvmsg` system call, and corresponds to the receive timestamp of the frame that carried the last byte returned by the system call. The LANCET application-parsing logic leverages this information to associate timestamps to replies of variable sizes: if the content returned by `recvmsg` consists of an incomplete reply, that timestamp is ignored; if the content contains the replies to multiple requests (which is possible because of TCP's streaming nature and coalescing in the socket layer), LANCET only considers the timestamp for the last reply returned in that call. The Linux kernel coalesces `sk_buffs` internally and keeps a single timestamp per `sk_buff` corresponding to the last arrival. Consequently, earlier received responses might appear to have later receive timestamps.



(a) $\bar{S} = 1\mu s$ (b) $\bar{S} = 10\mu s$ (c) $\bar{S} = 100\mu s$

Figure 4.6 – Latency ECDF for an M/D/1 model and three deterministic workloads at 20% load.

Our contribution to the kernel 4.19.4 [235] guarantees that each `recvmsg` system call will return the hardware timestamp that corresponds to the last byte read. Previously, this was only the case for software in-kernel timestamping.

4.6 Evaluation

Our evaluation aims to answer three fundamental questions: (i) how does LANCET compare with existing RPC load-generating tools such as Mutilate and Treadmill (ii) how does LANCET's self-correcting methodology work in practice (iii) how LANCET performs in characterizing a server's behavior across different loads.

We answer these questions using a methodology in which the server's execution time is explicitly controlled. Doing so enables comparing the client-side measurements made by the tools to an idealized queueing theoretic model. We leverage an RPC server with synthetic service times following well-known distributions. Specifically, we tried a fixed, an exponential, and a bimodal distribution where 10% of the requests take ~10× longer to execute. To further reduce server-side overheads, our server uses the open-source IX operating system [34] configured with 1 CPU and adaptive batching disabled. The operating system overhead is ~1µs of CPU execution time per request, which includes driver and network processing overheads. As baselines we use the opensource versions of Mutilate [1] and Treadmill [3]. For Treadmill, we had to make changes in order to build it for our setup.

To be able to compare with other tools, our synthetic server uses the `ascii-memcached` protocol. Clients submit `get` requests with for a 19-byte key (similarly to Facebook's USR [14]), the server spins for a configurable amount of time, and replies that that key was not found. We chose `ascii-memcached` because it is the only protocol supported by both Treadmill and Mutilate.

The idealized models correspond to the expected latency distribution, as determined by a discrete event simulation, assuming zero operating system overheads, zero network propagation

delays, and zero client-side measurement overheads.

For all of our experiments, we configure each client machine with 15 threads and 4 connections per thread with 1 outstanding request per connection. Also, we consider a Poisson inter-arrival distribution of requests.

4.6.1 Experimental setup

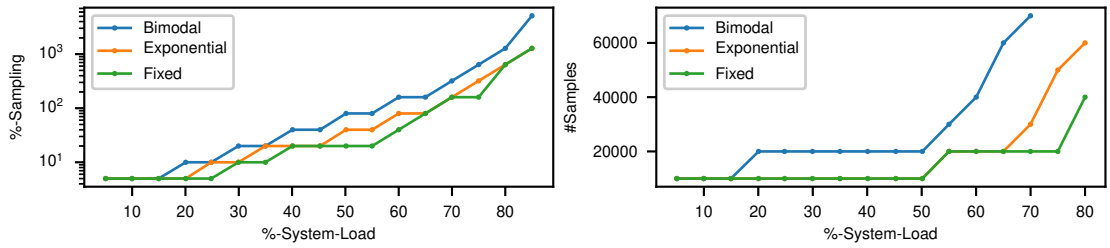
Our experimental setup uses 5 clients and one server machine connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The client machines are equipped with a Xeon E5-2637 @ 3.5 GHz and a Mellanox Connect-X4 NIC. The machines run an Ubuntu LTS 16.04 distribution running Linux kernel version 4.19.4. The systems are tuned to reduce jitter: all power management features, including CPU frequency governors and TurboBoost, and support for transparent huge pages, are disabled. The server is a Xeon E5-2665 @ 2.4 GHz with an Intel x520 NIC running the IX operating system.

4.6.2 Benefits of hardware timestamping

First, we compare the measuring granularity of LANCET with the measuring granularity achieved by Mutilate and Treadmill. For LANCET we consider both the hardware timestamping, symmetrical setup and the asymmetrical one based on the busy-polling agent. LANCET and Mutilate provide a way to run an experiment based on multiple machines, but for Treadmill there is no opensource coordinator script. Thus, we run one Treadmill instance on each client that contributes 1/5 of the load. Also, we modified Treadmill to save the collected latencies at the end of the experiment.

From a methodology perspective, we plot the latency CCDF for a deterministic service time distribution with different average service times. The load is set at 20% of the theoretical saturation, we range the average service time from $\bar{S} = 1\mu s$ to $\bar{S} = 100\mu s$. We collected 1M samples for each tool.

Figure 4.6 summarizes the experiment results. We observe that LANCET, for both configurations and in all three experiments, achieves lower measuring granularities when compared to the other tools because it reduces the client measuring overheads. Specifically, for $\bar{S} = 1\mu s$ hardware timestamping measures a 99th percentile tail of 14.1µs and the LANCET polling agent one of 27.3µs. Mutilate measures 40µs and Treadmill reports 63µs. Figure 4.6a also shows that Mutilate's line is not smooth because of the µs reported granularity, as opposed to nanoseconds reported by the other tools. Also, we see that LANCET aligns better with the theoretic results. For example, with $\bar{S} = 10\mu s$, the blue line nicely tracks the model; the offset between the two (~10µs) is essentially due to the operating system overhead and the propagation delay. Finally, Figure 4.6c shows that the tools make a difference even for coarser grain tasks ($\bar{S} = 100$), where the operating system and propagation delay overheads are comparatively small.



(a) Required level of sampling to guarantee *iid*-ness

(b) Required number of samples to achieve the target CI of 10µs

Figure 4.7 – Dynamics of LANCET's self-correcting methodology based on load for three service time distributions with $\bar{S} = 10\mu s$

For the rest of our evaluation we will focus on the symmetric hardware-timestamping agent as it reports the most accurate results.

4.6.3 LANCET self-controlling dynamics

In the next series of benchmarks, we want to identify the impact of the self-correcting methodology and how the coordinator controls the experiment parameters based on the different service time distributions and the system load. To do so, we run the three different service time distributions across a variety of loads and we collect the necessary level of sampling to achieve *iid*-ness, and the number of samples necessary for a target confidence interval size of 10µs.

Figure 4.7a shows the sampling rate that is necessary to the unbiased processing of the results caused by queuing effects. We observe that high-dispersion workloads (*e.g.*, bimodal) and higher load levels require lower sampling rates. This is expected as increasing either service time dispersion or load level leads to more queuing, thus more dependent samples.

In Figure 4.7b, we set the size of the target confidence interval for the 99th percentile latency to be 10µs with 95% confidence. The figure shows the number of collected samples, as decided by the coordinator, that are required to satisfy the result target. We observe that more samples are necessary to fulfill the constraint as the load increases, since higher system load leads to higher latency variability.

The bimodal distribution shows an interesting behavior of the tool: With load > 70%, execution stops after the maximum number iterations ($N = 10$) but the target CI expectations can not be met. Our experiment logs showed that the collected 99th percentile latency at 75% load is 411.333µs [-5.87µs, 7.56µs] at 95% confidence; this interval is > 10µs.

We also tested LANCET's self-correcting behavior with the lognormal distribution, which is a heavy tailed distribution. LANCET terminates without ever being able to confirm results convergence(CI < 10µs for the 99-th percentile latency), even at a low load of 20%. Thus,

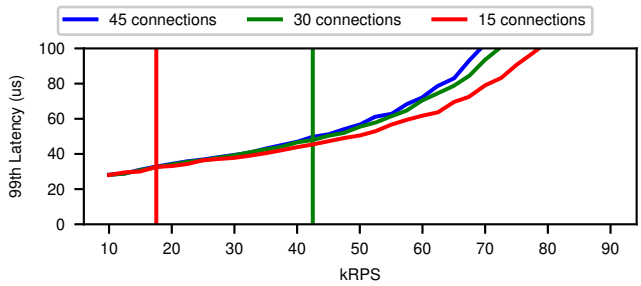
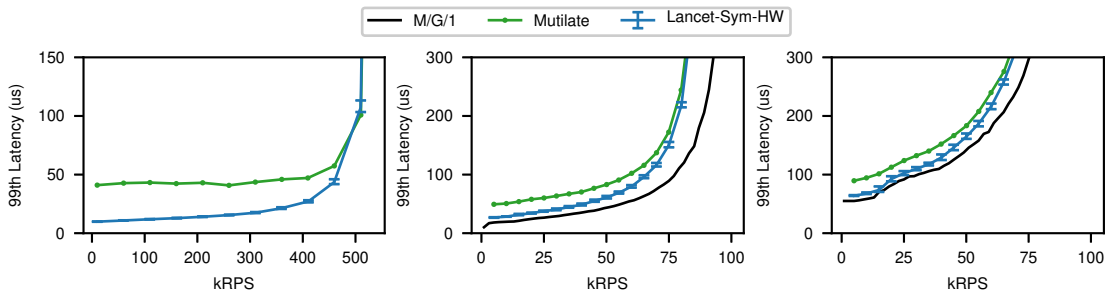


Figure 4.8 – Impact of the inter-arrival distribution to tail-latency for a fixed with $\bar{S} = 10\mu s$. Vertical lines correspond to the load levels where Lancet reports inter-arrival distribution violations



(a) Deterministic, $\bar{S} = 1\mu s$ (b) Deterministic, $\bar{S} = 10\mu s$ (c) Bimodal, $\bar{S} = 10\mu s$

Figure 4.9 – Latency vs throughput graphs for a 5-client experiment with average service time of $\bar{S} = 1$ and $\bar{S} = 10$

LANCET is effective in detecting heavy-tailed service time distributions.

4.6.4 Inter-Arrival distribution Impact

In the following experiment we try to showcase the impact of the inter-arrival distribution on the latency results and how LANCET identifies cases of inter-arrival distribution violations. We use the fixed synthetic time distribution with $\bar{S} = 10\mu s$ and we run a latency experiment across a variety of loads with different number of connections. We disable LANCET's checks for inter-arrival distribution and we only report whether there is a workload violation. To eliminate any system interference we configure LANCET with one connection per thread, and add connections by adding client machines.

Figure 4.8 shows the 99-th percentile latency as a function of throughput for the different connection count configurations. The vertical lines correspond to the load level that the equivalent configuration started violating the inter-arrival distribution. We observe that once LANCET reports a violation the curves start deviating. This experiment shows that cases as the one described in Figure 4.1 can be avoided by LANCET's self-correcting methodology.

4.6.5 Server characterization

Figure 4.9 shows the 99th-percentile tail latency as a function of the load for three workloads. We compare LANCET with Mutilate as well as the idealized, zero-overhead theoretical model. Both tools use 5 machines – necessary to achieve the high loads required. For LANCET, we additionally report the confidence interval of each measurement. This experiment does not include Treadmill as Treadmill's open-source distribution does not support multi-machine deployments. Note that because of system overheads, the IX server cannot get close to the expected maximum load for Figure 4.9a which would be 1M RPS, thus we do not plot the theoretic curve.

We observe that LANCET reports latencies that closely match the idealized model across the entire load spectrum, to the point that it accurately reflects the two inflection points of the binomial distribution. We also observe how the size of the confidence intervals change across different distributions and system loads. For low loads and low service time dispersion, the interval is shorter than the maximum configured (10µs). For the bimodal distribution, the reported confidence interval is at its maximum configuration even for low loads.

4.7 Related Work

LANCET is one of the many contributions towards enabling reproducibility and accurate experimentation in systems research [238, 169].

µs-scale computing: Recent research focuses on µs-scale computing [31] both in operating systems and networking and either aim to optimize [34, 290, 229, 299, 178], or attribute the sources of tail-latency [301, 223, 188, 379, 219]. LANCET does not attempt to attribute the sources of the jitter. Instead, it provides a tool to measure µs tail latency precisely on an end-to-end basis to be used in similar research efforts.

Precise measurements: RPC generators [379, 219, 114, 70, 249] use software timestamping. However, researchers need more accurate tools to evaluate system's latency, *e.g.,* ZygOS used a modified version of Mutilate based on DPDK [88], and MICA [229] used a custom version of YCSB on DPDK. Software-based packet generators [64, 98, 264] also used DPDK for increased precision [301]. Hardware-based packet generators [323, 163] provide sub µs-scale precision with little jitter [301]. Some tools repurposed the IEEE PTP feature of standard NICs to measure packet latencies [98, 252, 215]. LANCET is the first tool that leverages hardware-based NIC timestamping for capturing latency for RPCs over TCP with even higher precision. In addition, LANCET uses the standard Linux networking stack for all experiments, proving a more realistic simulation environment. While we used Mellanox ConnectX-4 NICs in our experiments, hardware timestamping of all packets is also available on Solarflare NICs [2].

Methodology: Although tail-latency is a widely used system metric, there is no widely accepted experiment methodology for measuring it, and usually tools are bounded to specific methodologies. LANCET attempts to split the methodology from the actual tool and reason about them separately. Measurement bias from non-determinism can be avoided via setup randomization [245, 127, 379]. Repeated runs eliminate hysteresis effects in systems [379]. Distributed benchmarking tools seek to minimize client side queuing bias by reducing the client load, in asymmetric *e.g.,* Mutilate [219], or symmetric setups [379]. LANCET's use of hardware timestamping eliminates client bias in the point of measurement. Treadmill [379] avoids issues of imbalance by leveraging a symmetric measurement model, and bias from outliers by computing interested metrics on individual instances and combining them using aggregation functions. LANCET also supports the symmetric setup to detect imbalance across client machines. Most tools use histograms to capture latencies. Treadmill determines bucket ranges during a calibration phase. YCBS [70] and Tailbench [188] have dynamic range histograms. LANCET relies on on-line sampling but keeps all sampled results to determine both the CCDF and the confidence intervals. Confidence intervals can also be used to determine statistical convergence of results [245, 127]. LANCET's self-correcting controller relies on statistical tests to ensure stability and results convergence similarly to [238].

4.8 Capter Summary

LANCET is a new latency-measuring tool designed with the explicit goal to accurately measure µs-scale tail-latencies while reducing methodological pitfalls in a principled manner. Its self-correcting methodology uses proven statistical methods to detect situations where application tail latency cannot be reliably measured. LANCET's agents uniquely leverage NIC-based timestamping to measure the end-to-end latency of TCP-based applications, completely eliminating client bias. LANCET measures latency distributions with more accuracy than popular tools such as Mutilate and Treadmill. Our evaluation with µs-scale workloads shows that it robustly self-corrects as a function of the load for workloads with challenging service time distributions.

5 Switch LOad Generation (SLOG)

5.1 Introduction

Today's webscale datacenter applications such as search, social networking, and e-commerce, communicate using Remote Procedure Calls (RPCs) in complex fan-in and fan-out patterns under strict µs-scale tail latency Service Level Objectives (SLOs). To support those applications, software and hardware-based Network Functions (NFs), such as Google's Maglev [96] need to process millions of packets per second.

Accurate and efficient load generators and latency measuring tools are crucial in designing high-performance datacenter applications and NFs. Those tools need to satisfy three main requirements: i) measuring latency accurately in a µs scale, without being part of the measuring loop. ii) generating load at a very high throughput to stress highly scalable applications, *e.g.,* based on kernel-bypassing. iii) accurately emulate deployment conditions be generating load with a realistic inter-arrival distribution, *e.g.,* Poisson arrival.

There is a plethora of load generators and latency measuring tools that target either NFs or RPC services. Those tools can be implemented in software and depend on software timestamping such as Mutilate [219], and YCSB [70]. Others, such as Lancet [201], and Moongen [98] are implemented in software but depend on hardware timestamping capabilities that can be found in modern NICs. Finally, there are tools that use custom hardware appliances, *e.g.,* FPGAs, such as Spirent [323] or IXIA to generate load and measure latency. There is a obvious trade-off between the accuracy and performance of hardware solutions, versus the flexibility and ease of use of the software ones. Software tools that depend on hardware timestamping try to balance this trade-off but they are still limited by the CPU processing capabilities and NIC line rates.

Emerging programmable switches, such as Barefoot Tofino [28], constitute a very appealing building block as they provide an adequate level of programmability in languages such as P4 [283] without compromising line rate performance. Such appliances change the way we design systems and allow pushing functionality, traditionally implemented at the end-hosts,

to the network. To do so they depend on a configurable set of match-action tables, dataplane registers that outlive packets, and a series of fixed function units.

In this work we propose SLOG (**S**witch **LO**ad **G**enerator), a programmable load generator and latency-measuring tool based on a programmable Tofino ASIC. SLOG leverages the programming capabilities and the fixed function units of Tofino to generate load and measure tail latency for both NFs and RPC services, while being able to generate a Poisson inter-arrival distribution. According to our knowledge, SLOG is the only hardware-based tool that is able to generate a randomized inter-arrival distribution, which is crucial for a realistic latency experiment.

5.2 Design

We design and build SLOG on top of a Barefoot Tofino ASIC, splitting the functionality between the control and the dataplane. In our design we had to answer five basic questions: i) How to use a programmable switch to generate and timestamp packets? ii) How to match the request with the reply timestamps? iii) How to maintain the latency distribution? iv) How to generate a Poisson arrival distribution? v) How can a user configure SLOG without knowing P4? We further describe our solution to each of these design questions.

Switch packet generation and timestamping: Tofino ASICs are equipped with a *packet generation engine* as a fixed function unit. The engine can be configured to generate packets either on specific types of events, or periodically based on a predefined packet format. It can send packets individually or in batches, up to 100Gbps per pipe. Generated packets are injected into the packet processing pipeline, allowing them to be processed and forwarded like regular incoming packets. We use the packet generation engine to generate packets and the pipeline logic to parametrize them.

The Tofino ASIC is also equipped with timestamping capabilities with ns-scale granularity both in the ingress and the egress pipeline. We collect the Tx timestamps in the egress pipeline, to avoid the queuing time in the switch buffers, and the Rx timestamps in the ingress pipeline.

Matching timestamps: To compute the end-to-end latency SLOG has to match Tx timestamps of outgoing packets with the Rx timestamps of packets coming from the device under test. Due to the limited dataplane memory Tx timestamps cannot be stored in the dataplane while waiting for Rx timestamp. Also, we don't want to keep the Tx timestamp in the packet payload, since the payload might need to be specific to the application under test. Thus, we decided to put the Tx timestamp in packet fields that are returned to the switch. Specifically, we can use the transport (UDP/TCP) source port to store 16 bits of the Tx timestamp. This field is returned as a destination port in the reply. Other fields that can be used to store the rest of Tx timestamp is the source IP, following a similar logic. Considering that the switch serves a hypothetical /16 subnet we can leverage the lower 16 bits of the source IP to store part of the

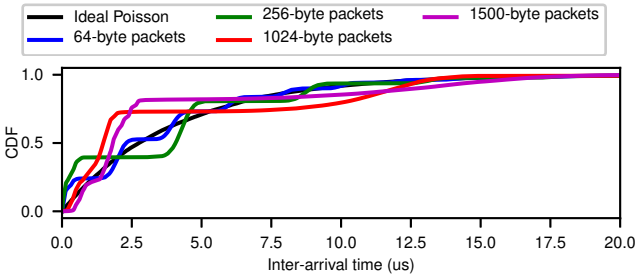


Figure 5.1 – Approximation of a 2MPPS Poisson process with different packet sizes observed at 1/8 target servers

Tx timestamp. There are also application specific fields that can be used for such purpose. For example, for DNS experiments we can use the *transaction id* to store another 16 bits of the Tx timestamp.

Latency distribution: After collecting the Tx and Rx timestamps for a specific request, the dataplane has to keep the latency sample till the end of the experiment to identify the latency distribution. However, maintaining all the latency samples in the dataplane is not feasible. So, we resort to *histograms*. We leverage table counters to implement histograms in the dataplane. Counters count the number of hits that each table entry gets. They are automatically updated in the dataplane every time there is a table hit as part of the match-action pipeline. Counters are only read in the control plane. At the beginning of each experiment, we statically generate a table that stores a sorted set of numbers at a target granularity, that correspond to the histogram bucket boundaries. After computing a latency sample, the dataplane tries to range match the value to the bucket table with a nop action. A successful match corresponds to an increase to the equivalent bucket. At the end of the experiment, the control plane dumps the content of the buckets to generate the latency histogram.

Inter-arrival distribution: Unfortunately, the packet generation engine can only generate packets either at a fixed rate or after a specific dataplane event. None of the above, though, fits our needs to generate a Poisson arrival distribution. Instead, we emulate this behavior by devising a mechanism to control the traffic rate in the dataplane. Rather than applying any inter-packet gap at the packet generator, we let it generate packets as fast as possible, and the dataplane transmits only a subset of the generated packets while dropping the rest.

Identifying the number of packets to be dropped is crucial for the inter-arrival distribution since it is translated to the inter-packet time interval. For a specific packet size, we compute the time the packet generator takes to generate a packet and inject it in the pipeline. This duration, $\Delta\tau$, is out granularity of control. At the beginning of the experiment the control plane generates a set of random inter-packet gaps according to the target inter-arrival distribution, transforms them to multiples of $\Delta\tau$, and stores them in a table to be accessible by the dataplane.

Challenge	Solution
Packet generation	Use Tofino's packet generation engine fixed unit
Packet timestamping	Use egress timestamp for Tx and ingress timestamp for Rx
Timestamp matching	Include Tx timestamp as part of the echoed packet headers
Latency statistics	Maintain a latency histogram in the P4 dataplane
Inter-arrival distribution	Generate packets at line rate and drop in the dataplane
User configuration	Configure the P4 dataplane using a YAML cfg file

Table 5.1 – Challenges and how we approach them in SLOG

Once the experiment starts, the dataplane randomly selects a number from this table, and drop the following N generated packets. To generate a random number in the dataplane we hash the current timestamp. Figure 5.1 shows how SLOG approximates a Poisson arrival distribution under different packet sizes for a target load of 2M PPS.

User configuration: Users can use SLOG without changes to the P4 dataplane. We provide a convenient way to describe a test setup, by writing a configuration file. The configuration file is then read by the control plane application to set up and run SLOG. The format used is YAML and the configuration consists mainly of key-value options and lists of such options. Examples of configuration parameters are: the target host IP and port, the number of packets to be sent, the target rate, and the inter-arrival distribution.

Table 5.1 summarises the above challenges and the solutions given by SLOG. Figure 5.2 depicts the SLOG P4 dataplane. In blue boxes you can see Tofino's fixed function units, the Packet Generation Engine, and the Packet Replication Engine. Once the packet enters the pipeline, the dataplane classifies it as an outgoing or an incoming packet based on the packet's intrinsic metadata. If the packet comes from the Packet Generation Engine, namely it is an outgoing packet, it follows the top path. It is replicated if necessay and forwarded to the egress pipeline. In the egress pipeline, the dataplane gets the current timestamp and puts it in the packet headers. Then it identifies whether the packet should be dropped according to the target inter-packet gap, and if so it drops the packet. Otherwise, it sends the packet out. If the packet is classified as incoming packet in ingress, the dataplane gets the current timestamp, computes the latency based on the Tx timestamp found in the packet headers, matches the latency on the histogram table to increment the bucket counter, and drops the packet.

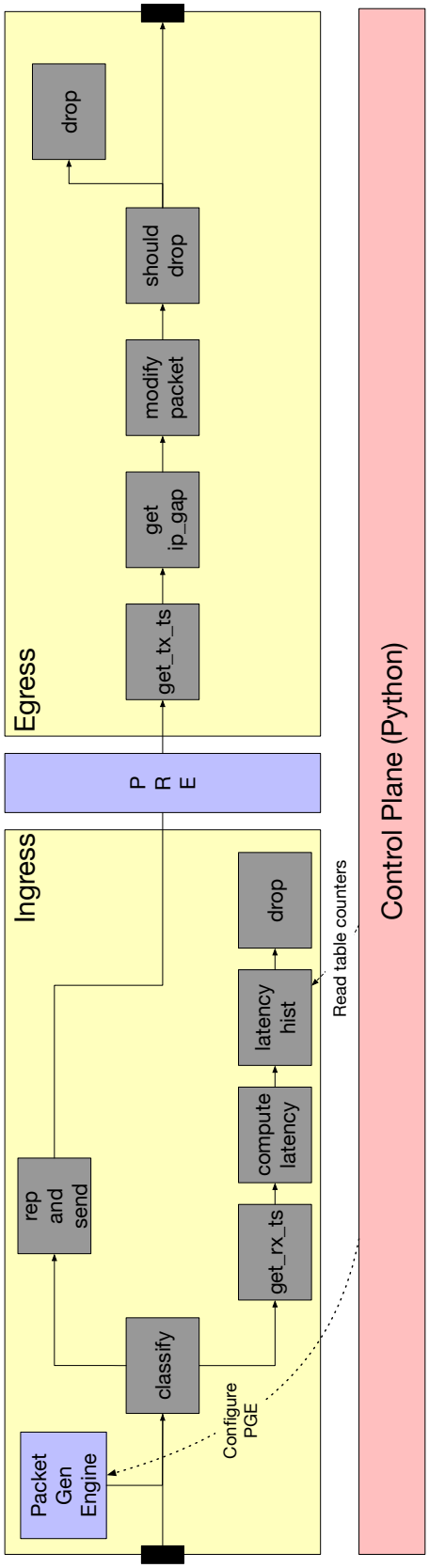


Figure 5.2 – The SLOG dataplane design

Part II

Latency-Critical Systems Using Existing Abstractions

6 Zygos: Achieving Low Tail Latency for Microsecond-scale Networked Tasks

6.1 Introduction

To meet service-level objectives (SLO), web-scale online data-intensive applications such as search, e-commerce, and social applications rely on the scale-out architectures of modern, warehouse-scale datacenters [29]. In such deployments, a single application can comprise of hundreds of software components, deployed on thousands of servers organized in multiple tiers and interconnected by commodity Ethernet switches. Such applications must support high concurrent connection counts and operate with user-facing SLO, often defined in terms of tail latency to meet business objectives [14, 80, 271]. To meet these objectives, most such applications distribute all critical data (*e.g.*, the social graph) in the memory of hundreds of data services, such as memory-resident transactional databases [337, 325, 358, 122, 361], NoSQL databases [304, 254], key-value stores [90, 229, 250, 376], or specialized graph stores [45].

These in-memory data services typically service requests from hundreds of application servers (high fan-in). Because each user request often involves hundreds of data services (high fan-out) and must wait for the laggard for completion, the SLO of the data services must consider the long tail of the latency distributions of requests [80]. Individual task sizes often require only a handful of µs of user-level execution each. These services would therefore ideally execute at the highest throughput, efficiently use all system resources (CPU, NIC, and memory), and deliver a tail-latency SLO that is only a small multiple of the typical task service time [31].

This hunt for the killer microseconds [31] requires researchers to revisit assumptions across the network and compute stacks, whose policies and implementations play a significant role in exacerbating the problem [219].

Our work focuses on the efficient scheduling on multicore systems of these very fine-grain in-memory services. The theoretical answer is well understood: (a) single-queue, multiple-processor models deliver lower tail latency than parallel single-queue, single-processor models and (b) FCFS delivers the best tail latency for low-dispersion tasks while processor sharing delivers superior results in high dispersion service time distributions [366].

The systems answer is, unfortunately, a lot less obvious, in particular when considering high request rates consisting of short messages and small processing times. In such situations, the state-of-the-art uses multi-queue NICs (*e.g.,* RSS [308]) to scale the networking stack across the multiple cores of the system. Current designs force users to choose between conventional operating systems (*i.e.,* typically Linux), and more specialized kernel-bypass approaches. The former can efficiently schedule the resources of a multi-core server and prioritize latency-sensitive tasks [55] but suffers from high overheads for µs-scale tasks. The latter improves throughput substantially (by up to 6× for key-value stores [34]) through sweeping simplifications such as separation of control from the dataplane execution, polling, run-to-completion, and synchronization-free, flow-consistent mapping of requests to cores [34, 290, 229, 167, 239, 183].

These sweeping simplifications lead to two related forms of inefficiencies: (a) the dataplane is not a work conserving scheduler, *i.e.,* a core may be idle while there are pending requests; and (b) the dataplane suffers from head-of-line blocking, *i.e.,* a request may be blocked until the previous tasks complete execution. While these limitations might be acceptable to workloads with near-deterministic task execution time and relatively loose SLO (*e.g.,* some widely-studied memcached workloads [250, 14] with an SLO at > 100× the mean service time [34]), such assumptions break down when considering more complex workloads, *e.g.,* in-memory transaction processing with a TPC-C-like mix of requests or with more aggressive SLO targets.

We present ZYGOS[1], a new approach to system software optimized for µs-scale, in-memory computing. ZYGOS implements a work-conserving scheduler free of any head-of-line blocking. While the design decisions voluntarily deviate from dataplane principles, ZYGOS retains the bulk of their performance advantages. The design, implementation, and evaluation of ZYGOS makes the following contributions:

(1) The design of ZYGOS, which leverages many conventional operating system building blocks such as the use of symmetric multiprocessing networking stacks, alternate use of polling and interrupts, inter-processor interrupts (IPI), and task stealing with the overall goal of delivering a work-conserving schedule. ZYGOS is architected into three distinct layers: (a) a lower networking layer, which runs in strict isolation on each core, (b) a middle *shuffle layer* which allows idle cores to aggressively steal pending events, and (c) an upper execution layer, which exposes a commutative API to applications for scalability [67]. The shuffle layer eliminates head-of-line-blocking while also offering strong ordering semantics of events associated with the same connection.

(2) The implementation of ZYGOS, which includes an idle loop logic designed to aggressively identify task stealing opportunities throughout the operating system and down to the NIC hardware queues. Our implementation leverages hardware virtualization and the Dune

[1] The Greek word for balancing scales.

framework [32] and handles IPIs in an exit-less manner similar to ELI [135].

(3) A methodology using microbenchmarks with synthetic service times to identify system overheads as a function of task size and distribution. This methodology allows us to identify both design limitations and implementation overheads. We apply this approach to Linux for event-driven execution models (using both partitioned and floating connections among threads), IX and ZYGOS and show that all converge as the task granularity increases, but at noticeably different rates, to distinct, well-understood models. For an SLO of 10× the mean service time at the 99th percentile, ZYGOS achieves 75%of the maximum possible theoretical load for 10µs tasks, and 88%of the equivalent load for 25µs tasks (Subsection 6.6.1).

(4) We compare ZYGOS to IX, a state-of-the-art dataplane with strict run-to-completion that partitions flows onto cores [34]. While ZYGOS's scheduler introduces some necessary buffering, communication and synchronization (which are measurable for extremely small tasks), it eliminates head-of-line blocking and clearly outperforms IX for tasks ≥10µs (Subsection 6.6.1). IX does outperform ZYGOS for workloads with very small task sizes such as memcached. The difference is primarily due to IX's adaptive bounded batching, which is not currently supported in ZYGOS. (Subsection 6.6.2)

(5) Last but not least, we evaluate the benefits of ZYGOS for an in-memory, transactional database running the TPC-C workload. Our setup uses Silo [337], a state-of-the-art, in-memory transactional database prototype. As Silo is only a library, we added client/server support to Silo, ported it to Linux, IX, and ZYGOS, and benchmarked it using an open-loop load generator for an SLO of 1000µs at the 99th percentile tail latency. ZYGOS can deliver a 1.63×speedup over Linux and a 1.26×speedup over IX. The speedup over Linux is explained by the use of many dataplane implementation principles in ZYGOS. The speedup over IX is explained by ZYGOS's work-conserving scheduler, which rebalances tasks to deliver consistently low tail latency nearly up to the point of saturation (Subsection 6.6.3).

The source code of ZYGOS, along with benchmarks, scripts and simulation models, is available in open source [384].

6.2 Queueing Analysis

There are at least three distinct forms of imbalance which impact tail latency that can be observed in systems:

1. Persistent imbalance occurs when different NIC queues observe different packet arrival rates over long intervals. Unless the system can share the load dynamically, some cores will be busier than others. This situation can occur if there is connection skew when some clients request substantially more data than the average, or if there is data access skew (*e.g.*, the CREW protocol in MICA balances reads but not writes across cores [229]).

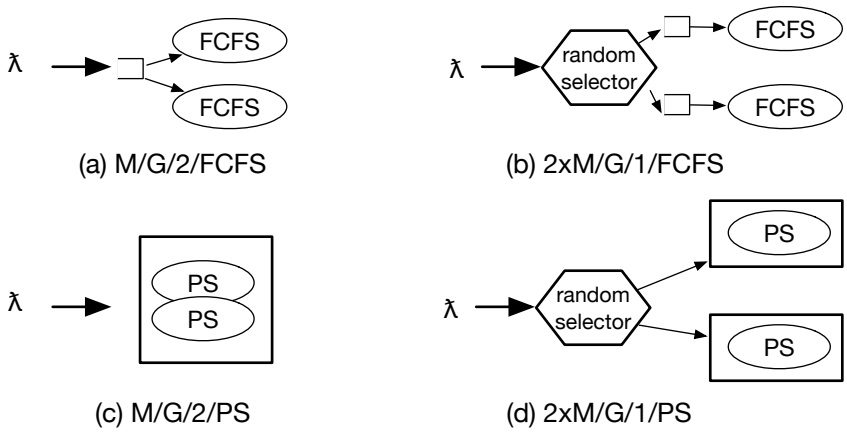


Figure 6.1 – Queuing models for $n = 2$ processors.

2. Arrival bursts cause temporary queuing even when the system is not saturated. The well-known Poisson arrival process has such bursts which cause the gradual increase in tail latency as a function of load, even if the time to process each request is fixed. In a multi-queue system, the Poisson arrival process generates bursts on different cores at different points in time. This creates a form of *temporary* imbalance that also impacts tail latency.

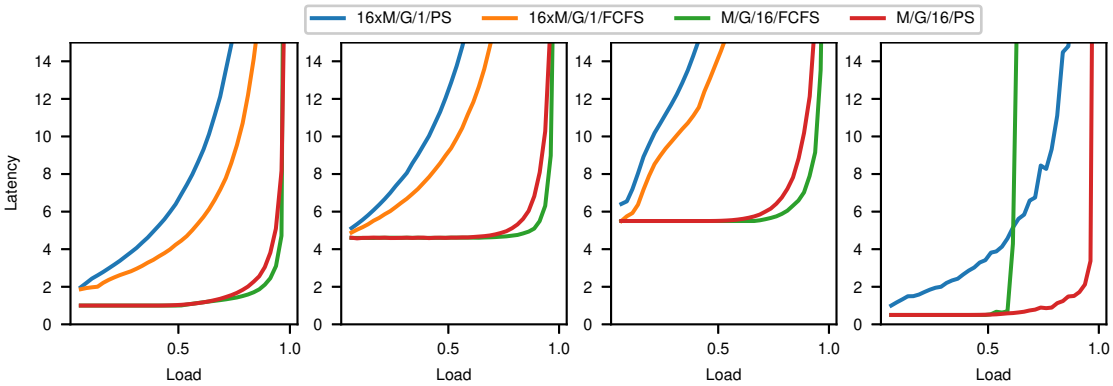
3. Service time variability will also create backlog and head-of-line blocking. A long request can occupy the processor for a long time, thus leading to a backlog of pending requests and a severe increase in tail latency.

We use four open-loop queuing models to build an intuition for the impact of arrival bursts and service time variability on tail latency. We use Kendall's notation to describe the models, where in the following expression `A/S/n/K`, `A` is the inter-arrival distribution, `S` is the service time distribution, `n` is the number of workers and `K` is the policy implemented, *i.e.,* first-come-first-serve (FCFS) or processor sharing (PS). For simplicity of the analysis, all models assume a Poisson arrival of requests (A=M). This is expected of many open-queuing models and representative of datacenter traffic with high fan-in connection counts. The Poisson process will generate arrival bursts and temporary imbalance in the multi-queue models, but no persistent imbalance.

Figure 6.1 illustrates the four different modes. Each delivers the same maximum throughput at saturation ($\lambda = n/\bar{S}$), but with different tail latencies due to different scheduling policies.

- The *centralized-FCFS* model (formally M/G/n/FCFS) idealizes event-driven applications that process events from a single queue or that float connections across cores (*e.g.,* using the `epoll` exclusive flag).

- The *partitioned-FCFS* model (formally n×M/G/1/FCFS) idealizes event-driven applications that partition connections among cores (*e.g.,* `libevent`-based applications) and



(a) Deterministic (b) Exponential (c) Bimodal-1 (d) Bimodal-2

Figure 6.2 – Simulation results for the 99th percentile tail latency for four service time distributions with $\bar{S} = 1$.

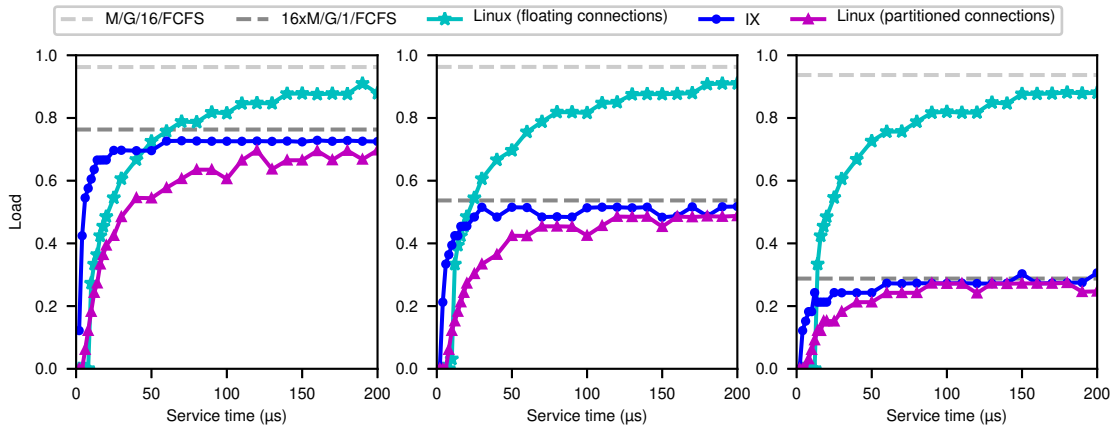
associate each core with its own private work queue. This model can be deployed on conventional operating systems or shared-nothing dataplanes

- M/G/n/PS idealizes the thread-oriented pattern (1 thread per connection) deployed on time-sharing operating systems. In practice, the task size granularity must be a multiple of the operating system time quantum.
- n×M/G/1/PS similarly idealizes the thread-oriented pattern when the operating system does not rebalance threads among cores.

Figure 6.2 illustrates simulation results for these idealized queueing models for a system with $n = 16$ processors. The figure shows the result for four well-known distributions [232]:

- deterministic $P[X = \bar{S}] = 1$
- exponential with mean service time $\bar{S}$
- bimodal-1:$P[X = \bar{S}/2] = .9; P[X = 5.5 \times \bar{S}] = .1$
- bimodal-2:$P[X = \bar{S}/2] = .999; P[X = 500.5 \times \bar{S}] = .001$

For each distribution, Figure 6.2 shows the tail request latency (queuing delay + service time) at the 99th percentile as a function of the load. Intuitively we understand that as the system load increases and approaches the system's limits, the number of requests in the queues also increases. That leads to an increase in the queueing time and tail latency. As expected, the minimum 99th-percentile latency is 1 for the deterministic distribution and 4.6 for the exponential distribution. As for the two bimodal distribution, b1 has a minimum tail latency of 5.5, which corresponds to the slow requests and b2 has a minimum tail latency of 0.5, which corresponds to its fast requests.



(a) Deterministic (b) Exponential (c) Bimodal-1

Figure 6.3 – Maximum load that meets the SLO as a function of the mean service time $\bar{S}$. The SLO is set at $\leq (10 \times \bar{S})$ at the 99th percentile. The grey lines correspond to the ideal upper bounds determined by the centralized-FCFS and partitioned-FCFS models.

We make two observations that inform our system design:

Observation 1: Single-queue systems (*i.e.,* `M/G/n/*`) perform better compared to systems with multiple queues (*i.e.,* $n\times$`M/G/1/*`): Systems with multiple queues, even with random assignment of events to queues, suffer from temporary load imbalance. This imbalance can create a backlog on some processors while other queues are empty. The lack of work conservation in such models limits performance. In contrast, single-queue models with a work-conserving scheduler (whether FCFS or PS) can immediately schedule the next task on any available processor.

Observation 2: FCFS performs better in regards to tail latency for distributions with low dispersion: This result has also been theoretically analyzed by Wierman et al. [366]. In Figure 6.2, FCFS outperforms PS for the deterministic, exponential and bimodal-1 distribution. PS only outperforms FCFS when the variance in service times increases, as in the case for bimodal-2. Note that partitioned-FCFS performs that poorly in bimodal-2 that is not obvious in these axis scales.

6.3 Experimental Methodology

We now describe the experimental methodology used to evaluate existing low-latency systems. The challenge is to define metrics that help determine (a) the inherent design tradeoffs by comparing real-life systems with the idealized models of Section 6.2; and (b) the sweet spot, in terms of mean service time and distribution, of each system. We use synthetic microbenchmarks to compare analytical results with experimental baseline results for three OS

configurations.

6.3.1 Approach and metrics

We rely on microbenchmarks with synthetic execution times to systematically compare different systems approaches for different task granularities. From the perspective of user-level execution, the applications are trivial: for each request, the application spins for an amount of time randomly selected to match both service time ($\bar{S}$) and distribution. From a systems perspective, the application follows the event-driven model to accept remote procedure calls sent over TCP/IP socket by client machines. The clients approximate an open-loop load-generator where incoming requests follow a Poisson arrival on randomly-selected connections [315]. All throughputs (requests per second) and 99th percentile tail latencies are measured on the client-side.

We use two metrics to compare systems: (a) the conventional "tail latency vs. throughput" is used to compare the efficiency of different systems for a given task granularity and distribution; (b) the "maximum load @ SLO" is used to compare the efficiency across timescales, for a given SLO expressed as a multiple of the mean service time.

This second metric is used to determine how fast different systems converge (or not) to the expected behavior of their idealized model, as the service time increases. For example, consider an SLO that requires 99% of requests to complete within $\leq 10 \times \bar{S}$. Queuing theory predicts a maximum load for each configuration, *e.g.,* for the exponential distribution a load of 53.7%for the partitioned-FCFS model and of 96.3%for centralized-FCFS.

6.3.2 Experimental Environment

Our experimental setup consists of a cluster of 11 clients and one server connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The client machines are a mix of Xeon E5-2637 @ 3.5 GHz and Xeon E5-2650 @ 2.6 GHz. The server is a Xeon E5-2665 @ 2.4 GHz with 8 cores (16 hyperthreads) and 256 GB of DRAM. All machines are configured with Intel x520 10GbE NICs (82599EB chipset). We connect the clients and the server to the switch through a single NIC port each. The client machines run mutilate [219] as a load generator: 10 machines generate load and the 11th one measures latency. The machines connect to the server over a total of 2752 TCP/IP connections. To minimize client latencies, we modified the latency-measurement agent of mutilate to use a DPDK-based, simple TCP/IP stack.

The machines run an Ubuntu LTS 16.04 distribution running Linux kernel version 4.11. Systems are tuned to reduce jitter: all power management features, including CPU frequency governors and TurboBoost, and support for transparent huge pages, are disabled.

6.3.3 Evaluated Systems

The synthetic microbenchmark models an event-oriented, scalable RPC server. During the setup phase, it accepts all connections from the client machines. During the benchmark, it simply receives request messages from the open connections, spins during the requested amount of time and returns a response. The server is setup as a 16-way multi-threaded application that uses all cores (and hyperthreads) and memory of the CPU socket connected to the NIC. We deliberately leave the other socket unused to eliminate the potential impact of NUMA policies in this study. We compare three configurations designed to support a large number of incoming connections:

- **Linux-partitioned:** This mode minimizes communication and application logic at the expense of load-imbalance: each thread accepts its set of connections (as directed by the RSS in the NIC [308]) and then polls on that same set during the benchmark. *Partitioned-FCFS* models the performance upper bound.
- **Linux-floating:** In this mode, all open connections are put into a single pool from which all threads may poll. Our implementation uses a simple locking protocol to serialize access to the same socket. *Centralized-FCFS* models the upper bound of performance.
- **IX:** The application uses the native dataplane ABI to receive socket events and respond correspondingly. This is also modeled as *Partitioned-FCFS*.

Linux configuration: The Linux systems were tuned to minimize latency and maximize throughput, by settling them on a configuration that limits the number of returned events by `epoll` to 1. We did observe that some of these settings had a surprisingly small, or even negative impact on either latency or throughput (*e.g.*, the `EPOLLEXCLUSIVE` commit evaluated the impact on thundering herds on a 250-thread setup whereas we only use one per core [100]). We attribute this to the fact that we pinned each application thread to a distinct core, thereby avoiding many of the subtle interactions associated with CPU scheduling.

IX configuration: IX can process bounded batches of packets to completion, which improves throughput only for very small task sizes. Unless when explicitly mentioned, we disabled it in our experiments as disabling batching noticeably improves tail latency. We also disabled the control plane and configured IX to use all 16 hardware threads of the socket and use the CPU at its nominal frequency of 2.4GHz.

6.3.4 Baseline results

Figure 6.3 shows the maximum load that meets the SLO of the 99th percentile $\leq 10 \times \bar{S}$ for three baseline operating system configurations described in Subsection 6.3.3. We include in greyscale two horizontal lines that correspond to the upper bound in performance, as

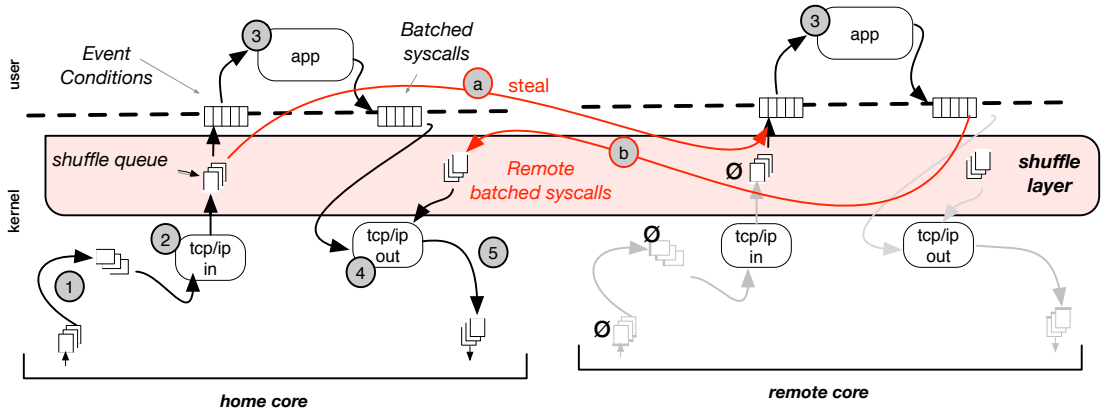


Figure 6.4 – Dataflow in the ZYGOS operating system. Steps (1) – (6) correspond to the normal execution on the home core. Steps (a)-(b) occur during stealing and involve the home and remote cores.

predicted by the *partitioned-FCFS* and *centralized-FCFS*, respectively. These upper bounds assume zero operating system overheads, no scheduling overheads, no propagation delays, no head-of-line blocking, no interrupt delays, *etc.* In addition, the centralized model assumes a perfect, global FCFS order of the allocation of requests to idle processors.

Figure 6.3 shows the result for three of the four distributions studied analytically in Figure 6.2. We omit the bimodal-2 results as the analysis of Section 6.2 showed that multi-queue systems have pathological tail latency with an FCFS scheduler. The figure shows clearly that:

(a) IX and `Linux-partitioned` both converge asymptotically to the expected 16×M/G/1 level of performance. Intuitively, we understand that as the service time increases, the overhead of the operating system becomes less prevalent. IX, which is optimized for small tasks, reaches 90% efficiency with tasks ≥25µs, ≥25µs, and ≥60µs for the deterministic, exponential, and bimodal-1 distributions. Larger tasks are required for Linux-partitioned to reach the same level of efficiency, *i.e.*, ≥120µs, ≥120µs, and ≥90µs, respectively.

(b) Yet, `Linux-floating` actually provides the best performance for larger tasks and slowly converges to the upper bound predicted by the centralized-FCFS model. The ability to rebalance tasks across cores allows it to outperform IX for tasks that are ≥50µs, ≥20µs, and ≥14µs for the deterministic, exponential and bimodal-1 distributions.

6.4 Design

6.4.1 Requirements

The theoretical analysis suggests, and in fact proves, that synchronization-free dataplane approaches cannot provide a robust solution to the tail latency problem, in particular when the service time distribution has a high dispersion. Yet, synchronization-free dataplanes

provide substantial throughput improvements over conventional operating systems.

We design ZYGOS, a single-address space operating system for the latency-sensitive data services, components of large-scale, online, data-intensive applications. Our design does not make any client-side assumptions or require any changes to the network protocol stack. We set the following hard requirements for our system design:

(1) Designed for current-generation datacenter architectures: Xeon multicore processors, 10GbE+ NICs with stateless offloads, Ethernet connectivity.

(2) Build a robust, multi-core, work-conserving scheduler free of head-of-line blocking for event-driven applications.

(3) Provide clean, ordering semantics of task-stealing operations to multi-threaded applications when handling back-to-back events for the same socket.

(4) Minimally degrade the throughput of short tasks when compared with state-of-the-art, shared-nothing dataplanes.

These hard requirements constrain the design space. While commodity operating systems such as Linux meet requirements #1 and #2, they provide only partial support for #3, which we will discuss in Subsection 6.4.3. As discussed in Section 6.3, the strict run-to-completion approach of dataplanes and their shared-nothing design is not an appropriate architectural foundation. We also rule out asymmetrical approaches which dedicate some cores to specific purposes (such as network processing) as the partitioning of resources is highly sensitive to assumptions on task granularities.

6.4.2 ZYGOS High-level Design

ZYGOS shares a number of architectural and implementation building blocks with IX [34]: each ZYGOS instance runs a single application in a single address space, and accesses the network through its dedicated NIC (physical or virtual function) with a dedicated IP address; each ZYGOS instance runs on top of the Dune framework [32]; a separate control plane can adjust resource allocations among instances.

Despite the lineage, ZYGOS is designed with radically different scheduling and communication principles than IX: IX is designed around a coherency-free execution model, *i.e.,* no cache-coherence traffic among cores is necessary, in the common case, to receive packets, open connections, or execute application tasks; ZYGOS is optimized for task stealing which has intrinsic communication requirements. IX achieves high throughput through adaptive batching, an approach that ensures that a batch of packets is first carried through the networking stack and then —without further buffering— processed by the application; ZYGOS uses intermediate buffering to enable stealing. Finally, IX is also designed around a run-to-completion model where it alternates execution between network processing and application

execution, which cannot be interrupted; ZYGOS relies on intermediate buffering and IPIs to eliminate head-of-line blocking.

ZYGOS achieves work-conservation with minimal throughput impact by architecturally separating the execution stack into three distinct layers, illustrated in Figure 6.4:

(1) the lower **networking layer** executes independently on each core, in a coherency-free manner. This includes the hardware/software driver layer, which relies on RSS to dispatch flow-consistent traffic to one receive queue per core. This also includes the layer-4 TCP/IP and UDP/IP layer, all of their associated data structures, intermingled queues, and timers. This design eliminates the need for any locking within the networking stack and ensures good cache locality.

(2) the intermediate **shuffle layer** introduces a new data structure per core: the *shuffle queue* is a single-producer, multiple-consumer queue which contains the list of *ready* connections originating from a given core. Connections in the shuffle queue contain at least one outstanding event and can be consumed by the core that produced it —the *home core*—, or atomically stolen by another *remote* core.

(3) the **application execution layer** manages the interactions between the kernel and the application through event conditions and batched system calls [321]. Each core has its own data structures and also operates in a coherency-free execution manner within that layer. Obviously, the application itself may have synchronization or shared-memory communication between cores and does not, in the general case, execute in a coherency-free manner.

Figure 6.4 shows the typical flow of events. Events numbered (1) – (5) occur when the packet is processed on its home core (*i.e.,* when no stealing occurs): (1) the driver dequeues packets from the hardware ring into a software queue; (2) the TCP/IP stack processes a batch of packets and enqueues ready connections into the shuffle queue; (3) the application execution layer dequeues the top entry, generates corresponding event conditions for the application and transfers execution to it. This, in turn, generates batched, system calls; (4) some system calls may call back into the network stack leading to execution of timers and/or (5) packet transmits. While the control flow resembles that of IX, the data flow is distinct as the shuffle queue breaks the run-to-completion assumptions as data is asynchronously produced into it and consumed from it.

Figure 6.4 also shows the interactions during a steal as the steps (a)-(b) in red. Consider the case where the remote core has no pending packets in the hardware queue, no pending packets in the software queue and no pending events in its shuffle queue. In step (a), it can then steal from another shuffle queue, which leads to the normal execution of the events in userspace, as step (3). The resulting batched system calls that relate to the networking stack are then enqueued for processing back at the home core in a multiple-producer, single-consumer queue, shown in step (b). Similar to the TCP input path, the TCP output path therefore also

executes in a coherency-free manner on the home core.

Figure 6.4 is only a high-level illustration of the system. In ZYGOS, each core is the home core of a set of flow-groups, as defined by the NIC RSS configuration and can act as the remote core for any other flow whenever it is idle. We now describe the ordering semantics that enable stealing (Subsection 6.4.3) and the data structures of the shuffle queue that eliminates head-of-line blocking (Subsection 6.4.4).

6.4.3 Ordering semantics in multi-threaded applications

When TCP sockets are statically assigned to threads, applications can rely on intuitive ordering and concurrency semantics [218]. The situation changes dramatically when sockets can float across cores as the `read` system call is not commutative when two threads access the same socket. Even though the Linux system call `epoll` allows it, and was even recently optimized for this use case [100], the implications on applications are far from trivial. Consider the case of back-to-back messages sent to the same socket (*e.g.,* two distinct RPC of the memcached protocol) for a multi-threaded application that uses the Linux-floating model of Subsection 6.3.3. Unless the application takes additional steps at user-level to synchronize across requests, race conditions lead to broken parsing of requests, out-of-order responses, or worse, intermingled responses on the wire. As a practical manner, applications or frameworks must, therefore, build their own synchronization and locking layer to eliminate these system races. This is sufficiently non-trivial that no known popular applications have done it to date, to the best of our knowledge. A related approach is the recent KCM kernel patch that provides a multiplexing layer of messages to TCP connections [192, 193].

With its goal to ensure very fine-grain work-stealing, we designed ZYGOS to free the application layer from the burden of synchronizing access to connection-oriented TCP/IP sockets. In this case, ZYGOS has an ownership model that ensures the events that relate to the same socket are implicitly ordered without the need for synchronization: whenever the home core or a remote core grabs an event for processing at the application layer, it grabs the exclusive access to the socket until the event execution has completed, including sending the replies on the TCP socket.

6.4.4 Eliminating Head-of-Line Blocking

The ordering semantics of Subsection 6.4.3 introduce a substantial complication to the design of the shuffle queue. ZYGOS eliminates head-of-line blocking by grouping events in the home core by socket. The shuffle queue has the ordered subset of sockets that are (a) not currently being processed on a core and (b) have pending data. The event queues are held in the per-socket protocol control block (PCB). While it offers strong ordering semantics to applications, this pre-sorting step does have an implication on the global order of packets, which is no longer guaranteed to be FCFS.

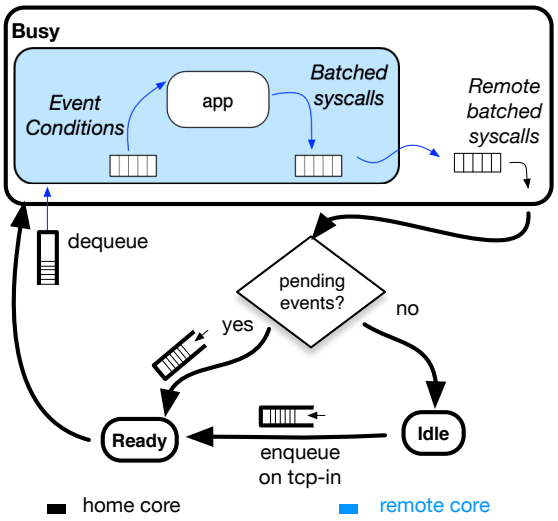


Figure 6.5 – Connection state machine transitions for the general case where an event is executed on a remote core (in blue). The connection is present in the shuffle queue exactly once when it is in the "ready" state, and never otherwise.

Figure 6.5 shows the state machine diagram that controls the decisions for each socket. Changes to the state machine and to the shuffle queue are atomic.

- *idle:* Sockets in this state have no pending incoming events, events currently processed by the application, or outgoing batched system calls.
- *ready:* The socket has pending incoming events, but is not currently being processed by the application and has no pending system calls.
- *busy:* The socket is associated with an execution core, which is either the home or remote core.

The execution core dequeues the first ready connection and creates the event conditions for the application. As previously discussed, system calls are returned back to the home core for processing. System calls may each generate asynchronous responses for that socket. After the execution of all system calls, the socket transitions either into the *idle* state if there is nothing further to process or into the *ready* state otherwise. In the latter case, the PCB is once-again enqueued into the shuffle queue.

6.4.5 Inter-processor Interrupts

The design in Subsection 6.4.4 eliminates head-of-line blocking concerns from the shuffle queue itself. In a purely cooperative implementation of ZYGOS, the cores poll on each other's data structures, which causes head-of-line blocking situations both before network processing as well as after application execution, since network processing explicitly takes place in the home core.

First, consider the case where packets are available for network processing in the hardware NIC queue but the shuffle queue is empty. This is the queue shown around step (1) in Figure 6.4.

As long as that core is executing application code, no remote core can steal the task. Idle cores poll both software and hardware remote packets queues. If pending packets exist, it sends an IPI to the remote core can force the execution of the networking stack, which replenishes the shuffle queue.

Second, remote batched system calls are enqueued by the remote core for execution on the home core (shown as step (b) of Fig 6.4). In a cooperative model, these system calls are only executed after the completion of application code, which unfortunately directly impacts RPC latency as some of these system calls write responses on the socket. Here also, an IPI ensures the timely execution of these remote system calls.

The shared IPI handler, therefore, performs two simple tasks when interrupting user-level execution: (1) process incoming packets if the shuffle queue is empty and (2) execute all remote system calls and transmit outgoing packets on the wire. The IPI interrupts only user-level execution since kernel processing is short and bounded. The kernel executes with interrupts disabled, thus avoiding starvation or reentrancy issues in the TCP/IP stack.

6.5 Implementation

The system architecture of ZYGOS is derived from the IX open-source release v1.0 [298]: it relies on hardware virtualization and the Dune framework [32] to host a protected operating system with direct access to VMX non-root mode ring 0 in the x86-64 architecture [339]. The kernel links in with DPDK [88] for NIC drivers and lwIP for TCP/IP [94]. The modifications to the application libraries are minor, but the kernel changes are extensive. Specifically, we modified ~2000 LOC of the IX kernel and ~200 LOC of Dune. While we retain the tight code base of IX, we revisit many of its fundamental design assumptions and principles.

The shuffle layer: We chose a simple implementation to ensure the atomic transitions described in Subsection 6.4.4. There is one spinlock per core which protects the shuffle queue of that core as well as the state machine transitions for sockets that call that core home. The lightweight nature of the operations that access it makes such a coarse-grain approach possible. Remote cores rely on `trylock` for their steal attempts to further reduce contention. Each PCB maintains a distinct event queue of pending events. This is a single-producer (the home core) and single consumer (the execution core) queue, implemented with one spinlock per PCB. The transitions from the *busy* state must test whether the PCB queue is empty and must first grab that lock.

Idle loop polling logic: The core design principle of ZYGOS is to ensure that an idle core will aggressively identify pending work. A core is idle when its shuffle queue, remote batch system call queue, and software raw packet queue are all empty. When it enters its idle mode, it starts to poll a sequence of memory locations, all of which are reads from cacheable locations. These locations include, in order of priority (a) the head of its own NIC hardware descriptor ring, (b)

the shuffle queue of all other cores, (c) the head of all unprocessed software packet queues of all other cores, and (d) the head of the NIC hardware descriptor of all other cores. For steps (b-c-d), the order of access is randomized. While heuristics could tradeoff a reduction of interrupts for a slight degree of non-work conservation, our current implementation aggressively sends interrupts as soon as a remote core detects a pending packet in the hardware queue and the home core is executing at user-level.

Exit-less Inter-processor Interrupts: ZYGOS relies on inter-processor interrupts to force a home core to process pending packets identified in steps (c) and (d) of the idle loop and to execute remote system calls back on the home core. Using an approach similar to ELI [135], we added support in Dune for exit-less interrupts in non-root mode, based on the assumption that ZYGOS kernel's interrupt handler will redirect to the Linux host operating system the interrupts that are destined to it. There is, however, no guarantee that the destination CPU will be VMX non-root mode when it receives the interrupt. We use interrupt 242, which is also used by KVM [195]. Interrupts received in root-mode are simply ignored by the KVM handler. As interrupts are used exclusively as hints, the unreliability of delivery impacts tail latency, but not correctness.

Control plane interactions: The IX control plane implement energy proportionality or workload consolidation by dynamically adjusting processor frequency and core allocation [300]. It operates in conjunction with the IX dataplane, which reprograms the NIC RSS settings. In principle, ZYGOS is compatible with these RSS settings changes, although policies and mechanisms would have to be adjusted as ZYGOS introduces new forms of buffering. We leave the evaluation of these interactions to future work.

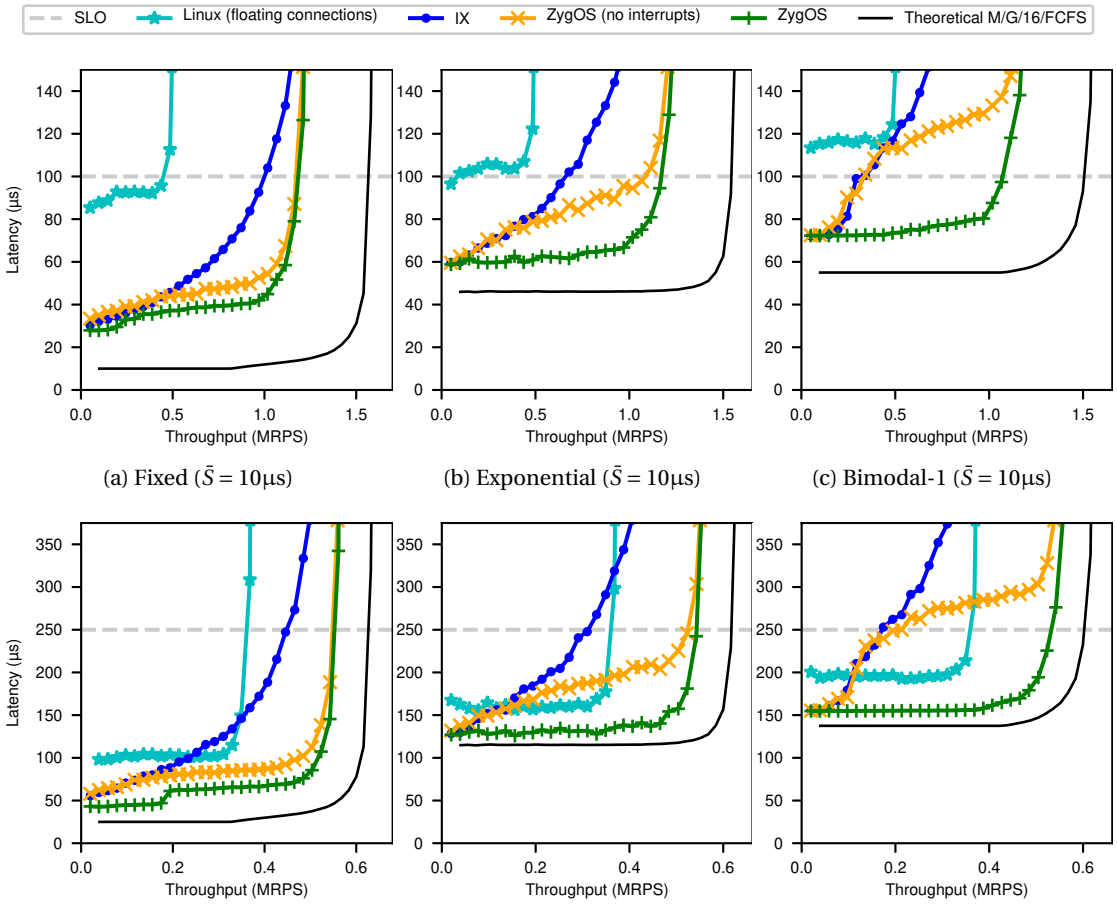
6.6 Evaluation

We use the same experimental setup explained in Section 6.3 to evaluate ZYGOS in a series of microbenchmarks, use memcached [250] to evaluate overheads on tiny tasks, and with a real application running TPC-C [336].

6.6.1 Synthetic micro-benchmarks

Figure 6.6 shows the latency vs. throughput of the three synthetic micro-benchmarks of Section 6.3. We compare ZYGOS with existing systems (IX and Linux) as well as the theoretical performance of a zero-overhead M/G/16/FCFS model for two granularities of interest, namely 10µs and 25µs. We observe that:

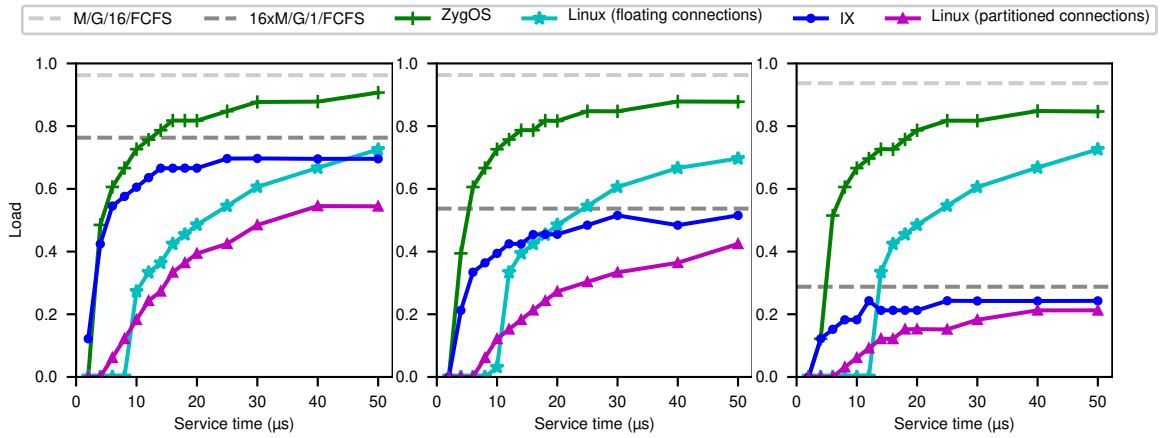
- ZYGOS and `Linux-floating` both approximate the theoretical model, with ZYGOS substantially reducing tail latency over IX;



(a) Fixed ($\bar{S} = 10\mu s$) (b) Exponential ($\bar{S} = 10\mu s$) (c) Bimodal-1 ($\bar{S} = 10\mu s$)

(d) Fixed ($\bar{S} = 25\mu s$) (e) Exponential ($\bar{S} = 25\mu s$) (f) Bimodal-1 ($\bar{S} = 25\mu s$)

Figure 6.6 – 99th percentile tail latency according to throughput for three distributions, each with 10µs and 25µs mean task granularity. The horizontal line corresponds to the SLO of $\leq 10 \times \bar{S}$.



(a) Deterministic (b) Exponential (c) Bimodal-1

Figure 6.7 – Maximum load that meets the SLO of the 99th percentile $\leq 10 \times \bar{S}$. The grey lines correspond to the ideal upper bounds of the two theoretical, zero-overheads, models (centralized-FCFS and partitioned-FCFS).

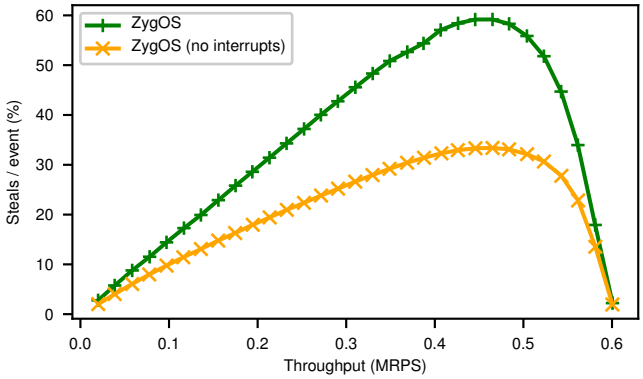


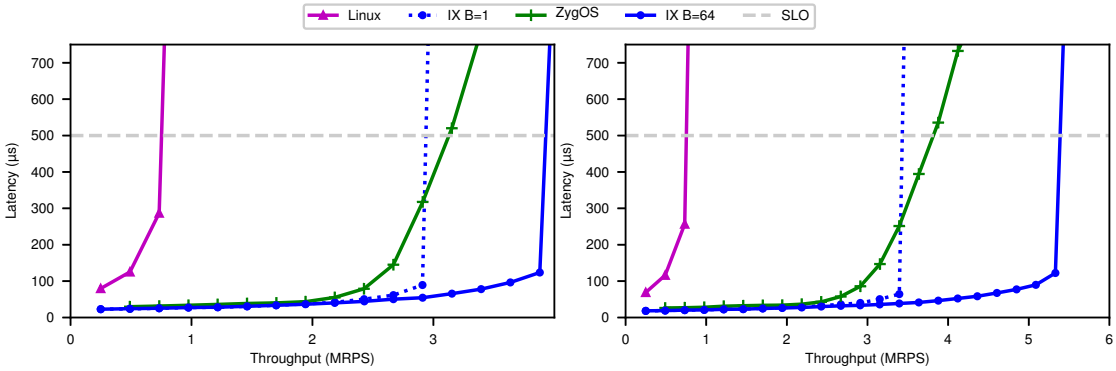
Figure 6.8 – Normalized rate of stealing vs throughput for exponential service time with mean 25 µs

- ZYGOS and IX have comparable throughput, even for tasks as small as 10µs; both clearly outperform Linux;
- for the exponential distribution, ZYGOS achieves 75%throughput efficiency at the SLO at $10 \times \bar{S}$ for $\bar{S} = 10\mu s$ (Figure 6.6b) and 88%for $\bar{S} = 25\mu s$ (Figure 6.6e);
- interrupts are necessary to eliminate head-of-line blocking with medium and high dispersion workloads, and the cooperative model of `Zygos-no-interrupts`, which is typical of pure user-level application, visibly impacts tail latency.

Efficiency for the $10 \times \bar{S}$ tail latency SLO: Figure 6.7 reports the efficiency (in terms of max load at SLO) as a function of task size. We compare ZYGOS with the baseline shown in Figure 6.3. We note the reduced X-axis truncated to 50µs for visibility; efficiency is stable beyond that point. ZYGOS clearly outperforms IX and Linux for any tasks sizes ≥5 µs and all three distributions for such a tight SLO. ZYGOS reaches 90% of the maximum possible load as determined by the zero-overhead *centralized-FCFS* theoretical model for tasks ≥30µs for the deterministic distribution, ≥40µs for exponential and ≥40µs for bimodal-1.

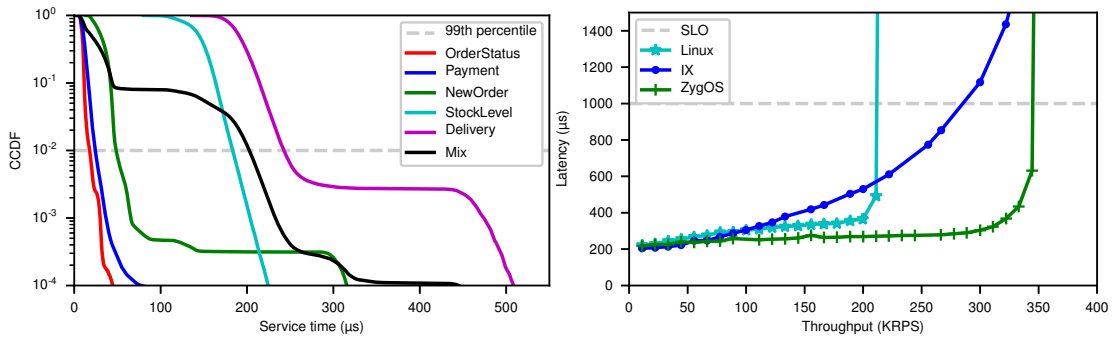
How much task stealing occurs?: Figure 6.8 provides an insight into the rate of stealing events as a function of load. The results are for the exponential distribution of Figure 6.6e but are remarkably similar for other distributions and timescales. As expected, there are few steals at low loads as more cores are near idle, and no steals at saturation as all cores are busy processing their own queue.

Without interrupts, temporary imbalances lead to a steal rate that peaks at ~33%. This rate is consistent with the peak of ~35% measured in a discrete event simulator that emulates the shuffle queue in a cooperative model without interrupts. Interrupts, —which are necessary to eliminate head-of-line blocking—, substantially increase the steal rate. At the peak, which corresponds to 77% of saturation, steals, and therefore interrupts are very frequent. Stealing



(a) ETC (b) USR

Figure 6.9 – 99th percentile tail latency vs. throughput for two memcached workloads for Linux, IX and, ZYGOS.



(a) Complementary CDF of task execution time (Linux) (b) 99th percentile end-to-end latency vs. throughput

Figure 6.10 – Silo running the TPC-C benchmark.

opportunities become less frequent as the load further increases.

6.6.2 Overheads of ZYGOS on tiny tasks: memcached

We compare the overheads of ZYGOS to IX for tiny tasks with the goal of identifying the task granularity where the sweeping simplifications of shared-nothing dataplanes such as IX noticeably improve throughput. We use memcached as an application ($< 2\mu s$ mean task size), and use the methodology and reproduce the results from IX [34]. We consider memcached a near worst case for ZYGOS as the application has very small task size with a small dispersion best approximated by a deterministic distribution.

Figure 6.9 shows the latency vs. throughput for the USR and ETC workloads, [14], as modelled by mutilate [219]. We compare Linux, ZYGOS, and IX. For IX, we choose two configurations: with adaptive batching disabled (B=1) and with adaptive batching enabled with the default setting (B=64).

First, we observe that ZYGOS and IX both clearly outperform Linux. We then note that for this particular SLO (500 µs), ZYGOS outperforms IX with batching disabled but lags behind IX with adaptive bounded batching. IX implements a strict run-to-completion model bounded by the batch size (B). ZYGOS currently implements adaptive bounded batching only on the receive path. It then processes events individually, interleaving between user and kernel code. While this hurts cache locality, it avoids head-of-line blocking. Similarly, it eagerly sends packets through the TX TCP/IP path and the NIC, also to avoid head-of-line blocking.

Of note, ZYGOS has a differently shaped latency vs throughput curve for this workload. As described in Subsection 6.4.3 and Subsection 6.4.4, ZYGOS does not respect strict FIFO ordering on servicing packets across different connections. For this workload configuration, up to four distinct memcached requests can be pipelined onto the same connection. The resulting reordering leads to a form of implicit batching of events, but only for those corresponding to the same flow. This implicit batching improves throughput but at an increase in tail latency. Such a behaviour is hard to restrict since ZYGOS doesn't know the boundaries of the requests in the TCP byte stream. Linux applications which use KCM sockets [192] can potentially handle this situation.

6.6.3 A real application: Silo running TPC-C

We validate the tail latency benefits of ZYGOS using Silo [337], a state-of-the-art in-memory database optimized for multicore scalability.

Application setup

Silo was originally implemented and evaluated as a library linked in with the benchmark. In the original evaluation, each thread runs as a closed loop issuing transaction requests, and in particular the TPC-C mix.

We ported Silo to run as a networked server accepting requests over sockets. We replaced the main loop of Silo with an event loop, which we used to run the workload on top of Linux, IX, and ZYGOS. The workload uses mutilate [219] with the same setup described in Subsection 6.3.2 to initiate transactions that then execute totally within the database server. Each remote procedure call generates one transaction from the TPC-C mix of requests.

We did not attempt to implement a marshalling of the full SQL queries and their responses, *e.g.,* over a JDBC-like protocol, as this falls outside the scope of the research question. We also note that Silo has a garbage-collection phase tied to its epoch-based commit protocol, which introduces a periodic barrier for all threads, with transaction latencies exceeding $1ms$. We disabled garbage collection for our measurements as it adds experimental variability, especially at the 99th percentile, depending on the experiment (and that taming the tail latency impact of Silo's GC also falls clearly outside the scope of this work)

System	Max load@SLO	Speedup	Tail Lat.@50%	Tail Lat.@75%	Tail Lat.@90%
Linux	211	1.00×	310µs (1.5×) @111	335µs (1.6×)@156	356µs (1.8×) @189
IX	267	1.26×	379µs (1.9×) @133	530µs (2.6×)@200	774µs (3.8×) @256
Zygos	344	1.63×	265µs (1.3×) @178	279µs (1.4×)@266	323µs (1.6×) @311

Table 6.1 – Maximum throughput under the SLO of 1000 µs and respective latencies at approximately 50%, 75%, and 90% of that load for each Silo running the TPC-C benchmark. Load is in KTPS. The number in the parentheses is the ratio of the 99th percentile end-to-end latency to Silo's 99th percentile service time (203µs).

Results

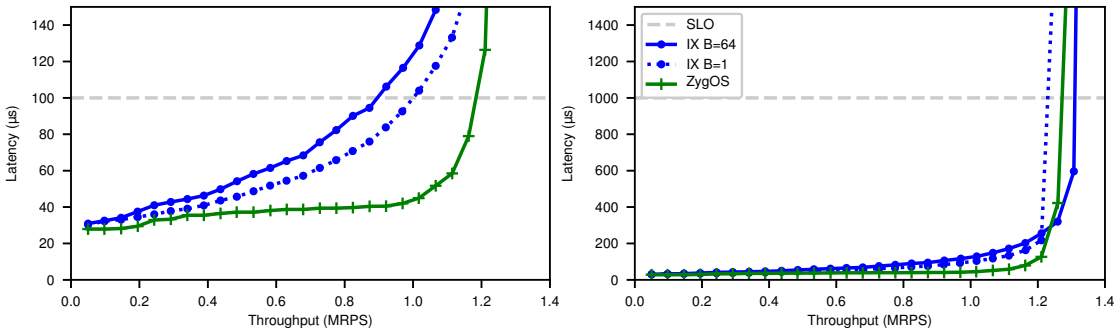
Figure 6.10a shows the complementary cumulative distribution of service time for the TPC-C benchmark for each of the five transaction types of the benchmark as well as the mix. The results were computed using Silo's master branch [319], with Silo locally driving the TPC-C benchmark. There is, therefore, no network activity, and indeed nearly no operating system activity. We run with GC disabled across all 16 hardware threads of a single CPU socket. The Figure reports the service time rather than that the end-to-end latency (*i.e.*, it excludes any queueing delays).

In this setup, the achieved transaction rate was 460 KTPS, which corresponds to the maximal throughput of the application, excluding any SLO and operating system overheads. Note that this TPS is consistent with the reported results in [337], given the differences in thread counts and processors. For the full mix, the average service time is 33µs, the median is 20µs, and the 99th percentile is 203µs. The figure clearly shows that Silo's service time distribution is overall multi-modal with small task granularity in the µs-scale.

Figure 6.10b shows the tail latency at the 99th percentile for Silo as a function of the load. To compare maximum loads, we selected a stringent SLO of 1000µs, which corresponds to ~33× the average and ~5× the 99th percentile tail latency. We observe:

- ZYGOS can support 344 KTPSwithout violating the SLO; this corresponds to a speedup of 1.63×over Linux. This demonstrates the benefits of our approach for real-life in-memory applications. The achieved transaction rate corresponds to 75% of the ideal, zero-overhead load with no SLO restrictions.
- This rate also corresponds to a speedup of 1.26×over IX. ZYGOS's work-conserving scheduler and its ability to rebalance requests across cores avoids SLO violations until the system becomes CPU bound on all cores.

Table 6.1 further quantifies the benefits of ZYGOS in terms of throughput at SLO and tail latency at a specific fraction of their respective maximum load. ZYGOS and Linux both deliver low end-to-end tail latencies for up to 90% of their respective capacity: 1.6×the 99th percentile service time for ZYGOS and 1.8×for Linux. This is anticipated by the *centralized-FCFS* model.



(a) 100 µs SLO

(b) 1000 µs SLO

Figure 6.11 – Comparison of IX (batch size 1 and 64) and ZYGOS for a deterministic service time of 10 µs and 2 different SLOs.

In contrast, as anticipated by the *partitioned-FCFS* model, IX delivers substantially higher tail latencies, *e.g.*, 1.9×when operating at half capacity, 2.6×at 75% capacity, and 3.8×at 90% capacity.

6.7 Discussion

6.7.1 The impact of SLO on systems

The choice of an SLO is driven by application requirements and scale, with the intuitive understanding that a more stringent SLO reduces the delivery capacity of the system. We show that the choice of an SLO also informs on the choice of the underlying operating system and scheduling strategy.

Figure 6.11 illustrates the tradeoff through the latency vs. throughput curves for the synthetic benchmark of Subsection 6.6.1 with an exponential service time of $\bar{S}$ =10 µs. Figure 6.11a and 6.11b actually show the results of the same experiment but on two different Y-axis corresponding to two different SLO. ZYGOS consistently shines on the more stringent SLO of 100µs (Figure 6.11a, $10 \times \bar{S}$) as the work-conserving scheduler tames the tail latency, followed by IX with batching disabled. For this SLO, IX (with batching enabled) consistently delivers the highest tail latency and violates the SLO with the lowest throughput.

However, for a more lenient SLO (Figure 6.11b, $100 \times \bar{S}$), IX's adaptive batching delivers marginally higher throughput than ZYGOS before violating the SLO.

6.7.2 Using Application Hints

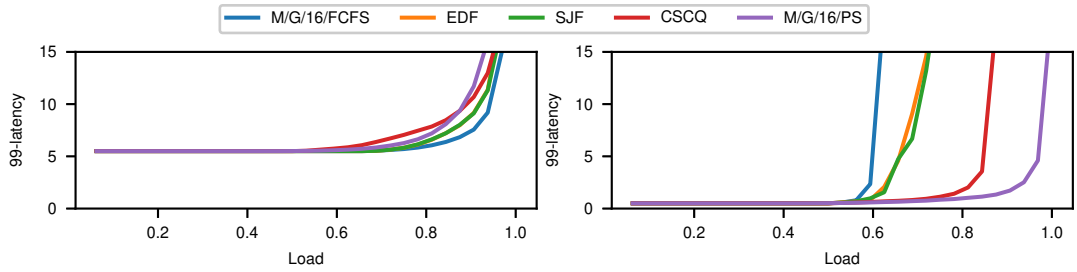
ZYGOS and systems that came after it, *e.g.*, Shinjuku [178], operate in a completely agnostic manner to the applications. All tasks have the same priority and their duration is unknown

before they start executing, thus they are treated equally. Consequently, both ZYGOS and Shinjuku have to make reactive scheduling decisions that depend on interrupts. ZYGOS uses interrupts to make sure that requests can be stolen from other cores after network processing, while Shinjuku uses interrupts to drive scheduling. IPIs though, complicate the system implementation, add extra overhead, and limit the environments where such systems could be deployed.

In this subsection we investigate the idea of using application-level knowledge to drive the scheduling decisions aiming to avoid preemptive scheduling, while still getting similar or even better tail latency results. The scheduling policies that take advantage of application-level information include Shortest-Job-First (SJF), Earliest-Deadline-Firs (EDF), and Cycle Stealing Central Queue (CSCQ) [149]. In SJF the request service time is known before the request starts executing. So, a SJF scheduler maintains a single ordered queue based on the request duration giving priority to short requests. SJF is optimal for average completion time and it is widely used in packet scheduling and congestion control algorithms, such as Homa [255], and pFabric [9] aiming to reduce the flow completion time. In EDF each request is accompanied by a deadline set by the client. The EDF scheduler also maintains an ordered central queue but this time the queue is ordered based on the request deadlines. Finally, CSCQ, unlike the previous policies that are work-conserving, is not a work-conserving policy. A CSCQ scheduler splits requests into groups based on their a priori known service time. For simplicity let's assume there are only two request types, fast and slow requests. For each request type CSCQ dedicates a set of cores, slow and fast cores. Fast cores only serve fast requests, while slow cores can serve both fast and slow requests, but prioritize slow ones. It becomes clear that there are cases where fast cores are idle while there is a backlog of slow requests to be processed by the slow cores, hence CSCQ is not conserving. This design choice though reduces head of line blocking since a long request cannot block a short one. The opposite can happen but does not significantly affect tail-latency.

We use our discrete event simulator to compare the above scheduling policies. We specifically consider the case of the two bimodal distributions in which there is a clear separation between the different request types. We compare the application-aware policies with the single queue FCFS and PS schedulers that are optimal for tail-latency in different scenarios: FCFS in low service time dispersion and PS in high service time dispersion. For the EDF scheduler we consider a deadline that is five times the request service time. For the CSCQ scheduler we split the cores according to the CPU needs of each request category considering cycle stealing, thus flooring the number of workers for fast requests and ceiling them for slow requests, resulting in 7 workers for fast requests and 9 workers for the slow ones.

Figure 6.12 summarises the simulation results for a 16-worker scenario. The case of Bimodal-1 is not that interesting and the application-aware policies perform close to the optimal FCFS. In Bimodal-2, though, we observe that application aware policies perform much better than the non-preemptive FCFS and approach PS which is optimal in this configuration but requires preemption. CSCQ specifically provides the best results since its non-work-conserving design



(a) Bimodal-1

(b) Bimodal-2

Figure 6.12 – Comparison between the FCFS and PS policies with policies that take advantage of application-level information: Earliest-Deadline-First(EDF), Shortest-Job-First(SJF), and Cycle Stealing Central Queue (CSCQ)

eliminates head of line blocking without requiring preemption.

Despite not evaluating these policies in ZYGOS because of our design requirements to remain application agnostic, we acknowledge the potential of using application-aware scheduling. Such approaches will require the introduction of new non-POSIX APIs so that the scheduler is aware of the different request types, but they are expected to significantly simplify the system implementation since they do not require preemption, thus removing the interrupt handling overhead and enabling the deployment of such latency-critical systems in other non-bare metal environments.

6.8 Related Work

Traditional event-driven models: This is the de-facto standard approach for online data-intensive services with high connection fan-in. On Linux, the use of the `epoll` has substantially improved system scalability. While `epoll` can be used in a floating model, and the recent `epoll-exclusive` eliminates thundering herds [100], applications must still rely on additional, complex synchronization to take advantage of the feature. ZYGOS delivers built-in, ordered semantics that guarantee that the replies from back-to-back remote procedure calls on the same socket will be returned in order. However, unlike the case of Affinity-accept [289] where each connection remains local to the core that accepted it, ZYGOS enables a connection to be served by any available core. Hanford et al. [144] investigated the impact of affinity on application throughput and proposed to distribute packet processing tasks across multiple CPU cores to improve CPU cache hit ratio. Although our work does not consider cache effects, we also conclude that strict request affinity can harm performance.

Traditional multi-threading model: Operating systems pre-emptive schedulers such as CFS [55], BVT [93] favor latency-sensitive tasks. Applications can benefit from multithreading to lower tail latency of completion of tasks when the granularity is a multiple of the scheduling quantum

and the distribution has a high dispersion.

Shared-nothing dataplanes architectures: Systems such as Arrakis [290], IX [34], mTCP [167], MICA [229], Seastar [316] and Sandstorm [239] bypass the kernel(via frameworks such as DPDK [88] or netmap [306]) and rely on NIC RSS to partition flows among cores. These shared-nothing architectures (at the system-level) with run-to-completion approaches completely eliminate the need to make scheduling decisions. These sweeping simplifications noticeably increase throughput but are oblivious to temporary imbalances across cores. MICA uses a client-side randomizing protocol (CREW or CRCW) to eliminate some causes for persistent imbalances among cores but does not address temporary imbalances. Decibel [262] and Reflex [196] are designed for storage disaggregation, depend on the shared-nothing assumption and similarly do not handle imbalance. ZYGOS is designed to eliminate such cases of imbalance though work-stealing. RAMcloud clients leverage RDMA hardware to bypass the kernel and communicate with a cluster of RAMcloud servers, with an asymmetric, push-based approach to task scheduling [281]. ZYGOS works with commodity Ethernet NICs and handles I/O and protocol processing symmetrically on all cores, with a pull-based, work-stealing scheme for task execution.

Work-stealing within applications: This commonly-used technique that has been mostly implemented either within the application or in a userspace run-time that runs on top of the operating system. Run-times such as Intel's Cilk++, Intel's C++ Threading Building Blocks (TBB), Java's Fork/Join Framework and OpenMP implement work-stealing schemes. Optimizing or building such run-times has also been studied intensely academically, *e.g.,* [58, 69, 87]. Statically mapping connections to cores can result in load imbalance in event-based programs and requires a solution at the library level [126, 375]. Recent focus on work stealing for latency-critical applications is at coarser timescales. [220, 370, 148]. The prior work largely targets applications with millisecond-scale task granularities that are easily accommodated by conventional operating systems. ZYGOS implements work-stealing within the operating system itself for network-driven to eliminate both persistent and temporary imbalances and is suitable for µs-scale tasks. As an operating system, ZYGOS's use of IPIs eliminates all cooperative multitasking assumptions between the threads.

Cluster-level work-stealing: Finally, load imbalance has been extensively studied at cluster-scale. Lu et al. [232] proposed a 2-level load balancing scheme based on the power of two to load balance traffic towards the front-end of cloud services. Sparrow [282] also relies on power-of-two choices for batch job scheduling. Google's Maglev [96] is a generic distributed network load balancer that leverages consistent hashing to load balance packets across the corresponding services.

6.9 Chapter Summary

We presented ZYGOS, a work-conserving operating system designed for latency-critical, in-memory applications with high connection fan-in, high requests rates, and short individual task execution times. ZYGOS applies some well-proven work-stealing ideas within the framework of an execution environment but avoids the fundamental limitations of dataplane designs with static partitioning of connections. We validate our ideas on a series of synthetic microbenchmarks (with known theoretical bounds) and with a state-of-the-art, in-memory transactional database. ZYGOS demonstrates that it is possible to schedule µs-scale tasks on multicore systems to deliver high throughout together with low tail latency, nearly up to the point of saturation.

7 CRAB: Bypassing the Load Balancer Without Regrets

7.1 Introduction

Load balancing is ubiquitous: nearly all applications today running in datacenters, public clouds, at the edge, or as core internet services rely on some form of load-balancing for both availability and scalability. Load balancing can have different forms, *e.g.*, L4, L7, DNS-based *etc.*, and can be implemented in hardware or in software. There has been considerable research on load balancing [275, 11, 96, 189, 286, 27, 251, 124, 292, 242] both from academia and industry due to not only the demands for mass deployments, high throughput, and low latency variability, but also the demands to lower provider resources specifically dedicated to it. For instance, Google reports that software-based load balancing can take up to 3-4% of a datacenter's resources [96].

This chapter focuses on internal load balancers, which are deployed between clients and servers within the same datacenter or public cloud. Internal load balancers can have a significant impact on the end-to-end latency both due to their load balancing decisions and the intermediate hop, while also constituting a major part of the infrastructure costs for cloud tenants. A common pattern includes the deployment of an internal cloud service,

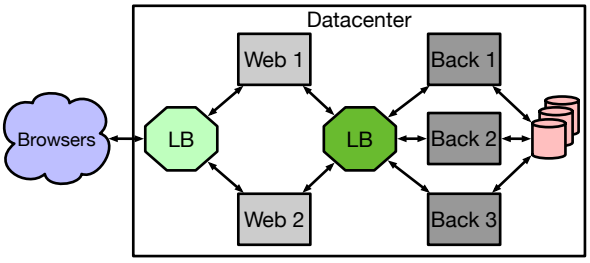


Figure 7.1 – Sample 2-tier cloud application. Web servers handle web traffic coming from users' browsers and act as clients for the back-end servers that run the application logic and communicate with a managed database. The light green octagon is an external load balancer while the dark green one is an internal load balancer.

placed behind an internal load balancer, that spawns new service instances according to load requirements and registers them with the load balancer, leading to seamless scalability and elasticity.

Figure 7.1 illustrates sample cloud-based, two-tier application. Users using their browsers hit the public IP of the external load balancer and their requests end up being served by the two web servers. Those servers act as internal clients for the backend-servers that are behind the internal load balancer and communicate with a managed database service. This design pattern allows the web tier and the back-end tier to scale independently and remain agnostic to each other due to the use of the two load balancers. Similar examples of such design patterns for services (or microservices) include ML inference to create recommendations, a user authentication microservice [123], generic application servers, and any workload orchestrated in containers such as Kubernetes[205].

Internal load balancers must be able to handle high-throughput, low-latency RPCs, typically implemented on protocols such as gRPC [139], Thrift [333], HTTP, or even custom protocols on top of TCP, *e.g.*, Redis, Memcache. The technical challenge is to spread the load as evenly as possible by leveraging rich, stateful scheduling policies while rapidly adjusting to changes in the service set, adding minimum latency overhead to the application, not creating I/O bottlenecks, and avoiding broken connections. State-of-the-art internal load balancers have benefited from recent innovation in protocol design specifically aimed at improving their scalability, including transport protocols other than TCP [179, 202, 255]. Such approaches though, break backwards compatibility with existing applications, while TCP still remains prevalent both for datacenter [7] and cloud communications. We note that this problem statement is different from that of external load balancers, who must accept and filter standards-based traffic from the Internet, mostly deal with HTTP(S) traffic and might also implement TLS termination.

Our approach *bypasses the load balancer without regret.* Specifically, we remove the load balancer from the critical path as much as possible and offer close to direct communication latencies. At the same time, our design allows for elaborate load balancing policies that improve tail-latency and quickly react to changes in the service set.

We design CRAB, a **C**onnection **R**edirect Lo**A**d **B**alancer. CRAB depends on a new TCP option included in the `SYN` and `SYN-ACK` packets that enables traffic redirection. This allows CRAB to only deal with `SYN` packets and stay off the connection datapath, thus tremendously reducing the load balancing load, while still being able to implement complex load balancing policies that otherwise would require a stateful load balancer implementation.

Our implementation shows that CRAB's datapath can be easily implemented using standard kernel-bypass mechanisms if running bare-metal, or as an eBPF filter when hosted in a VM for multi-tenancy purposes as a virtualized network function (VNF). The CRAB implementation in clients and servers requires a modest change; this can be implemented in a kernel module that has no measurable impact on performance or as direct kernel modifications offered as

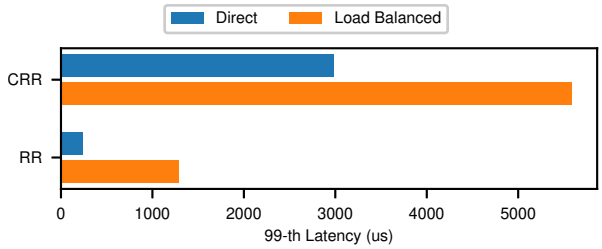


Figure 7.2 – Connection-Request-Response (CRR) and Request-Response RR latency benchmarks on Azure with accelerated networking with and without an Azure internal load balancer

pre-built images to cloud tenants.

Our evaluation demonstrates that CRAB outperforms L4-based load-balancers in terms of added latency overhead, connection throughput, and load balancing policies while being implemented on top of a simple stateless design.

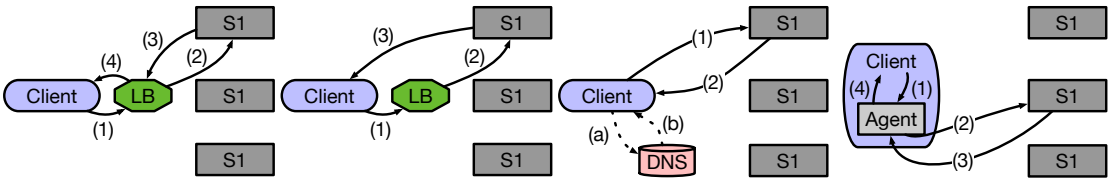
Our contributions are:

- The design of a backward-compatible extension to RFC 791 [295] that enables TCP connection redirection
- The design of a CRAB load balancer that depends on the new connection redirect feature of TCP and supports flexible scheduling policies.
- The implementation of the TCP connection redirection option in the Linux kernel for both clients and servers. Three implementations of the load balancer using DPDK, eBPF and Netfilter, respectively.
- A discussion on the caveats, assumptions, and opportunities for CRAB in the public cloud and for the integration of CRAB for Kubernetes NodePort load balancing.

7.2 Motivation and Background

In this section, we first showcase the problem we aim to solve and quantify the potential benefits CRAB can achieve. Then, we provide a comparative description of the state-of-the-art in load balancing that drives our design.

To begin, we run a simple experiment on the public cloud which mimics a scenario that many applications encounter today. We deploy two VMs on Microsoft Azure [19], one acting as a client and the other as a server both configured with accelerated networking [20]. Further, we place the server VM behind an Azure internal load balancer. In this setup, the client VM corresponds to the web tier and the server VM corresponds to the back-end tier from Figure 7.1. As benchmarks, we run a custom implementation of Netperf's CRR and RR benchmarks [348]. The CRR *(Connect-Request-Response)* benchmark measures the latency to open a connection,



(a) L4 Load Balancing (b) L4 Load Balancing with DSR (c) DNS Load Balancing (d) Agent-based Load Balancing

Figure 7.3 – Commonly used load balancing schemes for cloud services based on VMs or containers.

send an 8-byte payload, and wait for the server to echo the same payload. On receiving the response, the client closes the connection and starts over. In the RR *(Request-Response)* benchmark clients establish connections once and then use the same connection to send all their requests. RR measures the time between sending an 8-byte request and receiving the echoed back 8-byte response. Both experiments operate in a closed loop with one connection and one outstanding request at a time. Both the client and the server applications run on the vanilla kernel-based networking stack.

Figure 7.2 illustrates the $99th$ percentile observed latency for the CRR and RR experiments with and without the load balancer. Naturally, the latency for direct communication is lower than the load-balanced scenario. However, the latency overhead introduced by the load balancer is significant for both the RR and the CRR benchmarks. The load balancer adds approximately 1ms and 2ms respectively for the RR and CRR benchmarks; such a large overhead can overshadow the cost of non-balanced RPC.

Given this significant latency overhead associated with internal cloud load balancing, our goal is to minimize it as much as possible to achieve latencies that are close to direct communication. To do so, we need to understand the underlying load balancing mechanisms and policies.

7.2.1 Load Balancing Flavors

In this section, we categorize and compare the state-of-the-art approaches to load balancing for internal cloud workloads running on top of VMs or containers. Our comparison is based on the following criteria:

- **Load Balancing Policy**: Centralized policies leverage a global view that includes every back-end server while distributed policies make scheduling decisions based only on local state.

- **Persistent-Connection-Consistency (PCC) Violations**: Can the load-balancer route all packets from the same connection to the same back-end server in the presence of server arrivals and failures?

- **Expected Load**: What is the load balancer load in terms of the packets it has to process for each connection?

- **Latency Overhead**: How much overhead does the load balancer add?

- **Updates**: How quickly does the load balancer take into account scale-up (server-arrival) and scale-down (server-removal) events?

Layer 4 Load Balancing: L4 load balancers operate at the transport layer (TCP/UDP) of the networking stack and remain agnostic to the upper application layers. All public cloud providers offer some form of L4 load balancing, examples include Microsoft's Azure Load Balancer [22], which was used for the experiment in Figure 7.2, and Amazon's AWS Network Load Balancer [17]

Figure 7.3a describes the communication between the client, load balancer, and back-end servers for an L4 load balancer. The load balancer listens to a virtual IP (VIP) and the client uses this IP to talk to the service. The service is run on back-end servers that listen to some direct IP (DIP). The load balancer assigns each connection to a particular back-end server and performs address translation. It modifies the destination IP (to the DIP) for packets sent by the client and the source IP (to the VIP) for packets sent by the server. This requires all packets to go through the load balancer adding a latency overhead of 1 RTT to the end-to-end client-server communication and reducing the I/O scalability of the load balancer.

An optimization to the above approach is Direct Server Return (DSR). In this scheme, packets originating at the server can be sent directly to the client without being routed through the load balancer. Servers are aware that they are being load balanced and modify the source IP of outgoing packets to the VIP using address rewriting mechanisms such as `tc` [349]. DSR reduces the load balancer's load since it now only processes client packets and reduces the latency overhead to 0.5 RTT. Figure 7.3b illustrates an L4 load balancer with DSR enabled.

There has been significant research [275, 275, 11, 96, 189, 27, 286, 251, 124, 292, 242] on L4 load balancers. All these approaches can be split into two main categories depending on whether or not they store per-connection state.

Stateless load balancers [275, 11] typically depend on some form of consistent hashing [186] and daisy chaining to ensure that packets with the same 5-tuple will always be forwarded to the same DIP. Relying on hashing to distribute load enables them to eschew per-connection state leading to better performance and scalability. However, this approach has two main caveats. First, load balancing policies are limited to hashing, namely random load balancing; this leads to load imbalances especially when connections are skewed. Second, despite the use of daisy chaining there remain corner cases during server arrival and removal that lead to PCC violations [27].

Stateful load balancers maintain per connection state to correctly route each packet they

Method\Property	Policy	PCC violations	Expected load	Latency overhead	Updates
L4	Central	Possible*	every packet	1 RTT for every RTT	Fast
L4 w/ DSR	Central	Possible*	one way packets	1/2 RTT for every RTT	Fast
L7	Central	None	every packet	1 RTT for every RTT	Fast
DNS	Central	None	1 RPC every few connections	up to 1RTT per connection	Slow
Local Agent	Distributed	None	every packet	none	Slow
CRAB	Central	None	SYN packets	1/2 RTT for every new connection	Fast

* In stateless L4 load balancers

Table 7.1 – Feature comparison between different deployed load balancing schemes and CRAB.

receive from the client. Further, such load balancers can also maintain state about each back-end server, in order to support more elaborate load balancing policies such as *Join-Shortest-Queue* or Power of two [253]. Such policies cannot be implemented on a stateless load balancer. While per-connection state eliminates PCC violations, the state lookups can become a bottleneck when the number of active connections is large.

L7 Load Balancing: L7 load balancers or reverse-proxies operate at the application layer. These load balancers terminate client connections and open new connections to the back-end servers. Figure 7.3a could also describe a L7 load balancing scheme since all the received and transmitted packets have to go through the load balancer. However, for L7 load balancing arrows (1),(4) and (2),(3) would belong to different TCP connections. Popular open-source L7 load balancers include NGINX [269] and HAProxy [146]. Cloud providers also offer such services, *e.g.,* Amazon's AWS ALB [17].

L7 load balancers are typically centralized. Terminating client connections and establishing new ones with back-ends servers, enables them to avoid PCC violations. Further, operating at the application layer allows such load balancers to understand L7 protocols, *e.g.,* HTTP; this enables them to perform fine-grained request-level load balancing as opposed to the more coarse-grained connection level load balancing. However, this results in them depending on complicated software that typically run in userspace. This has the corresponding performance implications, in particular a considerable increase in the latency overhead (we illustrate this in Subsection 7.5.2).

DNS Load Balancing: Another form of load balancing used both in the public internet as well as by container orchestrators such as Docker Swarm [328], and Mesos [152], depends on DNS. DNS load balancing relies on the fact that most clients use the first IP address they receive for a domain after DNS resolution. Typically, the DNS server sends the list of IP addresses in a different order each time it responds to a new client, using the round-robin method. As a result, different clients direct their requests to different servers, effectively distributing the load across the server group. Figure 7.3c describes the client, server, and DNS server interactions for a DNS load balancing scheme. Steps (a)-(b) can be performed once for several connections

(1)-(2).

DNS load balancing, while centralized, is extremely coarse grained, since it only balances the load at a per-client granularity. Further, to avoid the repeated overhead of DNS resolution and reduce the load on the DNS server, clients cache DNS entries; once an entry is in the cache, clients and servers talk directly. Despite its performance benefits, caching can cause severe load imbalance issues. Since clients use the same target IP until the cached entry expires, the system cannot mitigate load imbalances during this period. Also, removing servers from the back-end pool becomes challenging and slow, since administrators have to wait until every possible TTL for the associated entries has expired. DNS load balancing though does not suffer from PCC violations since clients and servers communicate directly.

Local Load-balancing Agent: This load balancing scheme is used in Kubernetes [205]. In a Kubernetes cluster, every node that runs networked containers also runs a local agent responsible for the network configuration. This agent maintains a consistent view of the network by subscribing to state changes in the configuration service (`etcd` [105]). Each service that runs multiple containers and requires load balancing is associated with a specific ClusterIP. The local agent keeps track of the container IPs that run the specific service. Every time a client uses the ClusterIP, the local agent picks one of the target containers and performs address translation for each transmitted and received packet. Thus, while the client believes it is communicating with the ClusterIP, it is actually communicating with a container running the service. Figure 7.3d describes the above load balancing scheme and the interactions between the client, the server and the local agent.

The main benefit of a local agent is the ease of deployment, since it is integrated into the orchestrator rather than an external service. Placement and scaling decisions automatically update the load balancing decisions performed by this local agent, through pub-sub [107]. However, this approach suffers from three main problems: (1) load balancing decisions are performed in a distributed manner on every server, masking the benefits of smart placement, (2) packet rewriting is on the critical path for every packet sent and received, leading to an increase latency and CPU utilization (3) changes to the pool of target containers take longer to propagate through the use of pub-sub since all machines in the cluster have to receive the update; unlike the case in systems where the load balancing service is standalone and clients and servers are agnostic to it.

Table 7.1 summarises the above comparison and position CRAB in the design space among the same axis.

7.2.2 Load Balancing Policies

So far, we have categorized the different load balancing policies used in the above load balancing schemes based on two criteria: (1) whether they are implemented centrally (*e.g.*, L4 load balancers) or in a distributed manner (*e.g.*, load balancing based on a local agent);

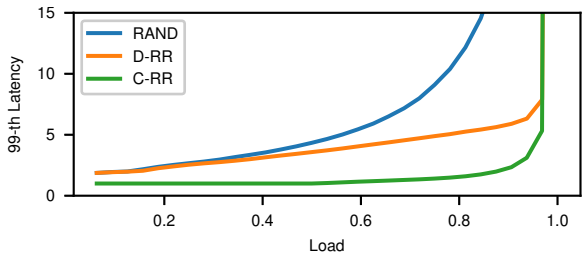


Figure 7.4 – Simulation result for tail latency to compare centralized and distributed policies.

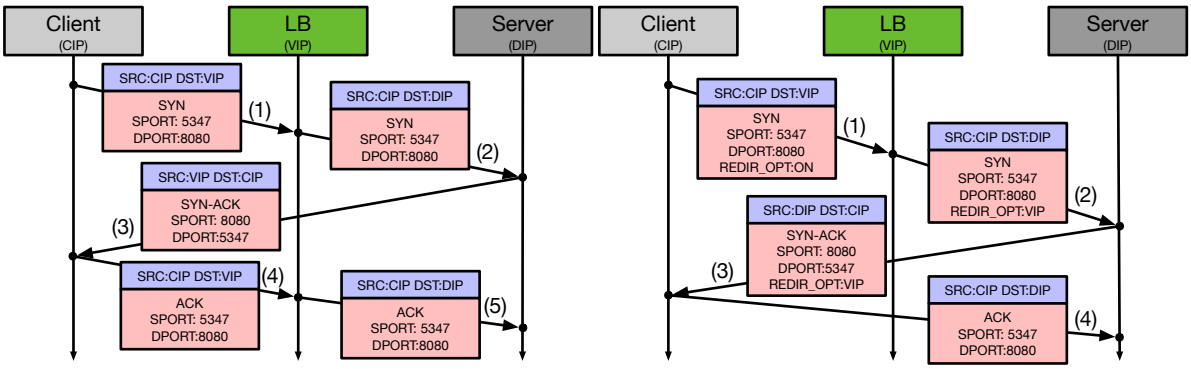
(2) whether the policy is random (*e.g.*, hashing in stateless L4 load balancers), or richer (*e.g.*, round-robin on stateful L4 load balancers). The performance implications of these policies on application tail latency, though, remains unclear.

To answer the above question, we leverage queuing-theory models and run a discrete event simulation for load balancing policies from each category. Our setup models 16 clients that communicate with 16 servers. Among stateful policies we choose the simplest non-random policy, which is Round-Robin. We simulate both the centralized and distributed versions of this policy. The centralized policy can be thought of as being implemented by a stateful L4 load balancers and the distributed policy as being implemented by Kubernetes-like local agents. Since stateless load balancing is equivalent to a random assignment of connections to servers, we simulate both centralized and distributed versions of the random policy. We exclude the case of the DNS-based load balancing due to the big impact of TTL both on the required DNS server resources, and the end-to-end latency.

Figure 7.4 summarizes the simulation results for each policy for a fixed service time distribution and Poisson arrivals. We observed that the centralized and distributed versions of the random policy displayed identical performance, hence we show them as one. In terms of application tail-latency, the random load balancing has the worst performance. Further, there is a difference between the D-RR (Distributed Round-Robin) policy and the C-RR (Centralized Round-Robin) policies. Due to the randomness of the Poisson process, performing Round-Robin on each node leads to worse tail latency, since centralized Round-Robin manages to always pick the least loaded server in this fixed service time experiment. We draw two conclusions from the above results: (1) Decentralized policies are worse at distributing the load than their centralized counterparts leading to worse tail latencies. (2) Rich policies can significantly outperform random load balancing.

However, the choice of load balancing policy is not independent of the load balancer design. For instance, richer policies require stateful implementations that do not scale well, while scalable stateless designs cannot support policies beyond random. This reveals a design trade-off between the scalability of stateless designs and richer load balancing policies of stateful designs.

Our goal is to design a load balancer that combines the best aspects of each design discussed so



(a) TCP handshake over a L4 load balancer with DST

(b) TCP handshake with connection redirection over CRAB

Figure 7.5 – A load balanced TCP handshake with and without connection redirection. Blue boxes correspond to IP headers, red boxes correspond to TCP headers.

far, namely: (1) the performance characteristics of DNS-based load balancing (expected load independent of flow size and close to direct communication latencies) (2) the load balancing capabilities of centralized, stateful L4 load balancers and (3) the scalability, flexibility, and PCC violation elimination of stateless L4 load balancers.

7.3 Design

CRAB is designed to satisfy the following requirements: (1) The load balancer must be able to implement centralized, stateful policies that offer better tail-latency, easier management, and faster updates. (2) The load balancer must not become an I/O and scalability bottleneck. (3) The load balancer must incur the minimal possible latency overhead by eliminating unnecessary network hops. (4) The load balancer must be backwards compatible with existing connection-based transport protocols (specifically TCP). (5) The load balancer must eliminate PCC violations.

The core insight behind CRAB is simple: *Implementing a centralized, stateful load balancing policy at a connection granularity requires the load balancer's involvement only during connection setup, following which the client and server can communicate directly.* Said differently, the load balancer need only map a connection to a back-end server when the connection is being setup, after which it only performs address translations, thus it can be taken off the data path. This eliminates all network hops through the load balancer in the data exchange phase, minimizing the latency overhead and avoiding scenarios where the load balancer becomes the I/O bottleneck, while it completely eliminates the risk of PCC violations. Note, this insight is specific to scenarios in which clients and servers can directly talk to each other and the load balancer is not required to conceal internal infrastructure. This assumption holds for internal load balancers in the public cloud (which is our target deployment) but does not generally hold for load balancers in the public internet.

CRAB realizes the above insight for TCP, by extending the traditional 3-way handshake. Figure 7.5a illustrates this handshake between a client and a server with an L4 DSR-enabled load balancer in the middle for a vanilla TCP implementation. This handshake requires 5 packets to be exchanged. The client first sends a TCP `SYN` packet to the load balancer's VIP (1). The load balancer assigns this connection to a particular back-end server and forwards the `SYN` packet to its DIP (2). Since DSR is enabled, the server replies directly to the client with a `SYN-ACK` packet having VIP as the source IP (3). Finally, the client sends the load balancer an `ACK`packet (4), which the load balancer forwards to the back-end server (5) to finish the connection establishment.

To remove the load balancer from the data path, CRAB leverages what we call *Connection Redirection* (CR). As the name suggests, CR enables redirecting the connection being established to a target IP address that is different from the one initially contacted by the client. To enable CR, we added a new TCP option called `Connection Redirect`. While clients would ordinarily discard `SYN-ACK` packets sent from an IP address they did not send a `SYN` packet to, they now conditionally accept such packets as long as they have the new TCP option. The `Connection Redirect` option includes a 4-byte field that carries the initial destination IP that the client sent the `SYN` packet to. This enables the client to validate that the received `SYN-ACK` is indeed a part of the handshake it initiated and also to find the associated `struct sock`. When a client receives a `SYN-ACK` with a valid `Connection Redirect` option, it changes its internal connection-related data structures and updates them with the new destination IP. Then, it sends the `ACK` to the new destination to finalize the connection establishment. Once this is done, the original destination IP is ignored and the two end-points communicate directly.

Figure 7.5b describes the TCP handshake, the associated packets, their IP and TCP headers, and other key fields in the case of connection redirection. As in the vanilla TCP case, the client first sends a TCP `SYN` packet to the load balancer (1). This TCP `SYN` now also indicates whether or not the client supports CR; in this case, we assume it does. The load balancer assigns this connection to a particular back-end server and forwards the `SYN` packet to its DIP (2). In addition, it uses the `Connection Redirect` option to include its VIP in the packet and inform the back-end server that this is a redirected connection. The server then sends the `SYN-ACK` packet directly to the client, with the source IP set to its own, and also echoes the `Connection Redirect` option with load balancer VIP (3). Finally, the client processes the new TCP option and redirects the connection, resulting in it sending the `ACK` packet directly to the back-end server and bypassing the load balancer (4).

Figure 7.6 illustrates the complete CRAB load balancing architecture. Steps $a - c$ correspond to the first 3 packets from Figure 7.5b. All further packets are directly exchanged between the client and the server, completely bypassing the load balancer. The only packets that the CRAB load balancer needs to handle are the TCP `SYN` packets. Once it directs the packet to a particular back-end server (load-balances the connection), it is eliminated from the data path. A direct consequence of this is that CRAB significantly reduces the resources necessary for load balancing. Since a CRAB load balancer handles only new connections, it can be provisioned

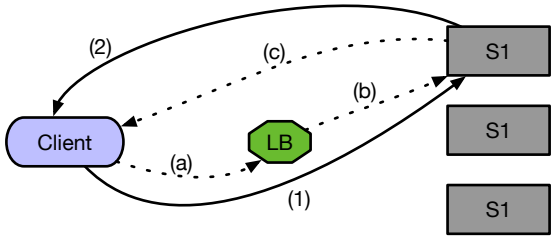


Figure 7.6 – Load Balancing over CRAB using the `Connection Redirect` TCP option. Dashed lines indicate connection establishment. Solid lines indicate data exchange.

according to the rate of new connection establishment, as opposed to being provisioned according to the receive and transmit throughput of those connections.

In cases where the client's `SYN` packet does indicate support for connection redirection, the load balancer can fall back to stateless hash-based load balancing, thus remaining compatible with non-CRAB-compliant clients. The load balancer check if the back-end servers are CRAB-compliant through the health probes already sent to make sure servers are up and running.

So, CRAB achieves its design goals as follows: (1) All `SYN` packets continue to be routed via the load balancer, allowing it to implement the centralized policy of its choice without the limitations of stateless load balancers. (2) Dealing with only `SYN` packets and not the actual connection payload, ensures that the load balancer is no longer the I/O bottleneck. (3) Removing the load balancer from the data path eliminates all intermediate network hops to it once connection establishment is complete. (4) CRAB is backwards compatible with existing network stacks and falls back to stateless load balancing if the `Connection Redirect` TCP option is not supported. (5) After connection establishment clients talk directly with servers, thus completely eliminating PCC violations.

7.4 Implementation

CRAB depends on a custom load balancing middlebox and requires changes to the client and the server endpoints. These three components can be implemented using different technologies based on deployment requirements, yet can inter-operate independent of the implementation. In this section, we describe the implementations in our current prototype. We discuss alternatives for both implementation and placement of this functionality in Section 7.6.

The deployment target for CRAB is a public cloud IaaS provider such as Amazon AWS, Microsoft Azure, or Google Compute Platform. We assume that the provider fully controls the physical infrastructure, but can also control the VM images that cloud tenants use. Unlike other deployment scenarios in which modifying the client endpoints is not feasible, *e.g.*, the internet, clients running on cloud infrastructure can easily integrate new features by running VM images offered by the cloud provider. This approach of specially modified could VM images is

not new and already used extensively, *e.g.*, in Azure accelerated networking [21].

7.4.1 Load Balancing Middlebox

We implement the CRAB middlebox in three different ways keeping in mind the infrastructure IaaS providers use today and the fact that they might need to run several load balancer instances per tenant. Consequently, we avoided an implementation based on programmable switches [28] and P4 [39] despite the fact that it provides the lowest latency and highest throughput. Instead, we built CRAB middleboxes that rely on DPDK [88], eBPF [344], and Netfilter modules [347] respectively.

Our DPDK-based CRAB implementation depends on a custom, simple networking stack and the load balancer implementation consists of ~100 lines of C code. Our eBPF-based load balancer leverages XDP [353] and runs natively in the driver context, thus avoiding an extra softIRQ. It processes the incoming packets and sends them out again through the same interface without letting them enter the Linux kernel, while also being able to easily communicate with the userspace through the use of eBPF maps that define the set of DIPs. The eBPF-based implementation consists of ~250 lines of C code. Finally, our Netfilter implementation runs in the context of a `NF_INET_PRE_ROUTING` hook. It is loaded as a kernel module and can communicate with userspace through a character driver to configure the target DIPs and the load balancing policy. The Netfilter implementation is ~200 lines of C code.

Our prototype implementations support two push-based load balancing policies namely random selection and Round-Robin. All implementations currently assume that clients and servers are CRAB compliant; they handle only TCP `SYN` packets and drop all other packets.

7.4.2 Connection Redirection

Server-side: To enable Conection Redirection, the server must include `Connection Redirect` option with the load balancer's VIP in the header of the `SYN-ACK` packet it sends to the client. This can be implemented either inside the kernel TCP stack or as part of a header rewriting mechanism before the packet is sent. We provide two implementations for this functionality that display similar performance characteristics. The first is based on a patch to the Linux kernel. It parses the TCP options in the received `SYN` packet and if the `Connection Redirect` option is set, echoes it in the `SYN-ACK` packet. The second implementation leverages Netfilter modules and hooks onto the `NF_INET_LOCAL_OUT` hook. This Netfilter hook modifies the outgoing `SYN-ACK` packets that match a certain source IP and port number and adds the `Connection Redirect` option with a predefined load balancer VIP. The Netfilter implementation totals ~200 lines of C. Given that the server-side implementation of CRAB does not require any kernel data structure modifications it can also be implemented in an eBPF program.

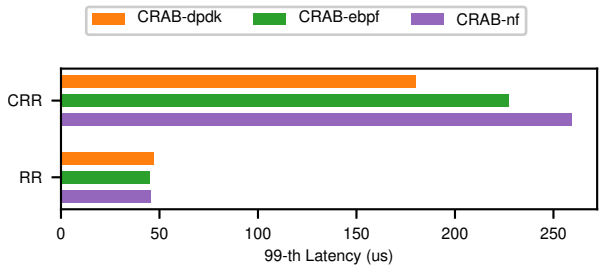


Figure 7.7 – Unloaded tail latency latency measured for the CRR and RR benchmarks for each CRAB load balancer implementation.

Client-side: The client-side of CRAB is the most intrusive since clients need to modify kernel data structures associated with the connection being redirected. We provide two solutions, one based on patching the kernel and another based on Netfilter modules; both display similar performance. Both implementations use the IP address found in the `Connection Redirect` option to locate the original connection and overwrite the connection's destination IP with the source IP of the received `SYN-ACK`. The Netfilter module totals ~150 lines of C and uses a `NF_INET_PRE_ROUTING` hook to modify the socket structure before the `SYN-ACK` reaches the TCP stack which otherwise would drop the packet due to the source IP being unknown. The kernel patch supporting CRAB is based on Linux 4.19.114 and adds ~200 lines of code.

7.5 Evaluation

Our evaluation answers the following questions: (1) How do the different implementations of the CRAB load balancer perform? (2) How does the latency overhead of CRAB compare against existing baselines? (3) How does system throughput scale with CRAB? (4) Can CRAB implement complex scheduling policies that improve the end-to-end application tail latency?

We evaluated CRAB on our infrastructure, rather than the public cloud due to limitations imposed by IaaS providers (*e.g.*, the inability to spoof IPs) which are necessary for the CRAB load balancer. Doing so enables us to be in full control and understand the infrastructure to better reason about the observed performance.

Our experimental setup consists of 10 machines connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The machines are a mix of Xeon E5-2637 @ 3.5 GHz with 8 cores (16 hyper-threads), and Xeon E5-2650 @ 2.6 GHz with 16 cores (32 hyper-threads). All machines are configured with Intel x520 10GbE NICs (82599EB chipset). The machines configured as clients or servers run either our CRAB-enabled modified Linux kernel or the CRAB client and server Netfilter modules since we did not observe any performance difference between the two implementations.

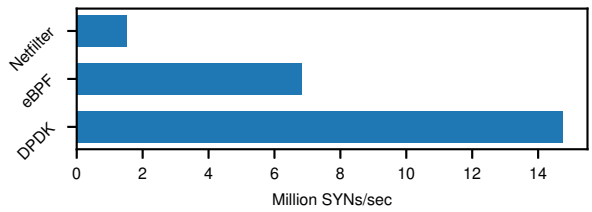


Figure 7.8 – Maximum throughput (new connections per second) achieved by each CRAB load balancer implementation.

7.5.1 CRAB Load Balancer Implementations

In this section, we evaluate the different CRAB load balancer implementations based on DPDK, eBPF, and Netfilter modules respectively, in terms of their latency overhead and throughput they can sustain. The goal is not to compare the performance of the three frameworks (DPDK vs eBPF vs Netfilter) but rather to provide information on the raw performance each implementation can achieve.

To evaluate latency, we run the same echo benchmarks as in Figure 7.2 with an 8-byte message size. In the CRR (Connection-Request-Response) benchmark, the client opens a new connection, sends a request, and waits for the response. On receiving the response, it closes the connection. We measure the end-to-end latency from connection establishment until the reply. In the RR (Request-Response) benchmark the client uses a pre-established connection to send requests and get back replies. We measure the end-to-end latency from when the client sends the request until the reply. Both experiments run in a closed loop with one client and one server thread, one connection, and one request at a time.

Figure 7.7 plots the 99-th percentile latency for the CRR and RR benchmarks for each implementation. As expected, we only observe a difference in the CRR experiment since the RR experiment uses a pre-established connection and the load balancer is off the data path. In the CRR experiment, the DPDK based implementation has the best performance since the load balancer works in a polling mode. The eBPF load balancer operates in native mode in the driver context, thus it performs better than the Netfilter module that runs in the softIRQ context and uses the kernel networking stack.

To evaluate throughput, we need to identify how many SYN packets can each implementation sustain since this is the only traffic that the CRAB load balancer deals with. Unfortunately, we do not possess the resources to run a full setup with TCP clients and servers such that the load balancer is the bottleneck. Instead, we created a DPDK-based client program that bombards the load balancer with SYN packets to stress test the different CRAB implementations. We configured the load balancers to redirect those SYN packets back to the same IP as if the client IP is one of the server DIPs. This enables the client to measure the throughput in terms of SYN packets per second, that each load balancer implementation can sustain. We configure the three load balancers to use only one core for packet processing. For the DPDK implementation,

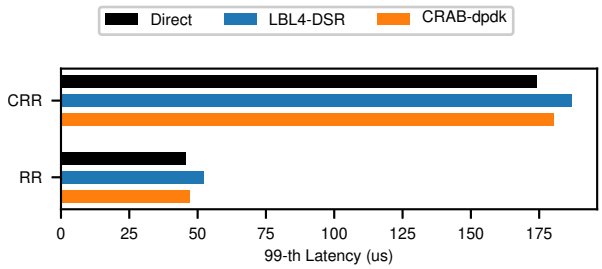


Figure 7.9 – Unloaded tail latency for the CRR and RR benchmarks for setups with no load balancer (Direct), a DPDK-based L4 load balancer with DSR, and the CRAB implementation on DPDK.

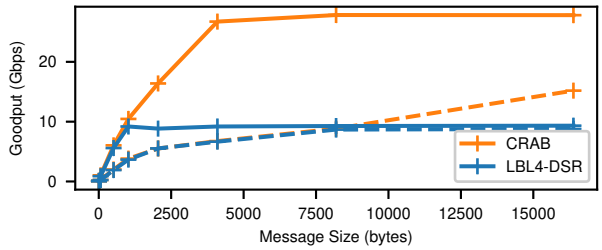


Figure 7.10 – Comparing Max goodput achieved in the CRR (dashed) and RR (solid) benchmarks for CRAB and a DPDK based L4 LB with DSR. I/O capacity of both LBs = 10G. I/O capacity of servers = 30G.

we run a single thread, while for the kernel-based implementations we redirected all NIC interrupts to one core, so packet processing would only take place on that core.

Figure 7.8 summarizes the results and plots the maximum achieved throughput in terms of SYN packets per second. CRAB-DPDK saturates a 10G NIC (~14M SYN packets per second) with a single core. CRAB-EBPF can serve 6.8M SYN/sec, while CRAB-Netfilter can serve 1.5M SYN/sec. As for latency, the performance difference is explained by the different interrupt contexts.

Given these results, we use only the DPDK-based implementation of CRAB from here onwards.

7.5.2 Latency Overhead

We now compare the latency overhead imposed by CRAB against existing load balancer implementations using the same unloaded latency benchmarks. We use the following baselines: (1) A direct configuration where clients and servers communicate directly without a load balancer. This represents the lower bound on latency. (2) NGINX configured as a TCP reverse proxy. This represents an L7 load balancer and (3) An implementation of a stateless L4 load balancer based on the Toeplitz hash [352] which runs on DPDK and is configured with DSR (LBL4-DSR).

Figure 7.9 plots the observed 99th percentile unloaded latency over 100000 samples for the direct, LBL4-DSR and CRAB load balancers. The NGINX configuration added ~250µs on top of direct for CRR, and ~50µs for RR and hence is not shown in the graph for visibility reasons. In the CRR benchmark, the DPDK implementation of CRAB adds ~6µs on top of Direct which corresponds to the half RTT overhead incurred when the SYN packet is routed through the load balancer. In comparison, the LBL4-DSR configuration adds double the overhead (~12µs) since both the SYN and the request packet from the client are routed through the load balancer. In the RR benchmark, CRAB performs the same as Direct, since the load balancer is off the data path. In contrast, the LBL4-DSR load balancer adds 6µs since the request packet from the client is routed through the load balancer.

7.5.3 Throughput Scaling

Here, we illustrate how CRAB's design enables the applications to bypass load balancer bottlenecks and scale their throughput to the I/O capacity of the back-end servers. As mentioned in Section 7.2, while DSR alleviates these bottlenecks for applications that are Tx heavy, CRAB seeks to eliminate this bottleneck for a broader range of applications. In this experiment, we target internal cloud services that have a symmetric throughput profile (*e.g.*, storage services, authentication/prediction microservices *etc.*).

To illustrate how CRAB enables throughput scaling, we run the same closed-loop echo benchmark in CRR and RR mode with different message sizes, but measure the maximum goodput (bytes of application payload per unit time) as opposed to the unloaded latency. We use three client and three server machines, and one machine serving as a load balancer, each machine configured with a 10G NIC. We compare CRAB against LBL4-DSR. Both load balancers implement the random load balancing policy.

Figure 7.10 illustrates the results. For both the RR and CRR workloads, the setup with L4LB-DSR is only able to achieve a maximum goodput of 10G since it is bottlenecked by the single machine that runs the load balancer. On the other hand, we see that CRAB can scale throughput beyond the capacity of the load balancer. In the RR experiment, it can achieve the goodput equal to the I/O capacity of the 3 backend servers ($3x10 = 30$G) at a payload size of 4096B. In the CRR experiment CRAB and LBL4-DSR perform similarly until a payload size of 8192B; this is due to the experiment being bottlenecked by the cost of new connection establishments. Past 8192B however, LBL4-DSR hits the 10G limit, while CRAB continues to scale.

7.5.4 Load Balancing Policies

So far, we've shown how CRAB, by redirecting connections and bypassing the load balancer incurs a lower latency overhead and enables better throughput scaling. We now evaluate whether CRAB supports elaborate load balancing policies that can significantly improve application tail latency, while still not maintaining per-connection state. Note that CRAB does not

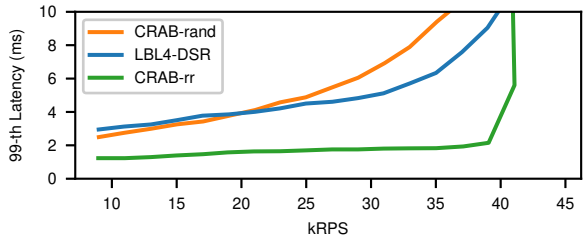


Figure 7.11 – Load Balancing 48 single-core servers running a synthetic service time application with $\bar{S} = 1$ms

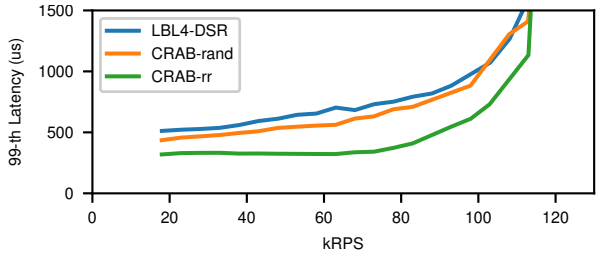


Figure 7.12 – Load Balancing 48 NGINX servers serving an 8 kB static file.

propose any new scheduling policies, but must support policies beyond stateless (random) load balancing. For now, we only evaluate push-based policies in which there is no explicit communication between the servers and the load balancer that could help the load balancing decisions. We leave more complicated policies to future work.

To run this experiment on our infrastructure but with a setup that resembles the public cloud, we use 3 servers and configure 16 virtual functions on each server for a total of 48 independent endpoints. We configure each VF with a unique DIP and equally rate limit the VFs so that they take a fair share of the 10G I/O bandwidth of the server. We evaluate the load balancing capabilities of CRAB for two classes of applications — CPU-intensive and I/O-intensive. The CPU bottlenecked application is a synthetic service time server with a fixed service time of 1ms. For the I/O-bottlenecked application, we use NGINX that serves a static file of 8kB over HTTP. For both applications, each request is sent over a new TCP connection. We use Lancet [201] as the load generator. Note, that in both experiments the load balancer is not the bottleneck.

We measure the tail latency as a function of application load for three load balancing policies: (1) Random load balancing with DSR — this represents policies supported by stateless L4 load balancers today (2) A CRAB implementation of random load balancing and (3) A CRAB implementation of Round-Robin load balancing — this represents richer load balancing policies that can only be implemented on stateful load balancers. CRAB, though, can implement Round-Robin without keeping per-connection state at the load balancer. The goal of the experiment is to validate if CRAB can realize the benefits of the elaborate policies as shown in Subsection 7.2.2.

Figures 7.11, 7.12 plot the tail latency vs throughput curves for the CPU bound and I/O bound applications respectively. We observe that for both application classes, despite all three policies achieving the same throughput, CRAB Round-Robin achieves significantly lower tail-latency. For application profiles with low service time dispersion, the Round-Robin load balancing policy picks the least loaded server and forward requests to it without requiring explicit communication between the load balancer and the server. Thus, CRAB in addition to eliminating I/O bottlenecks and reducing communication latencies, supports elaborate load balancing policies, truly achieving the best of all worlds.

7.6 Discussion

Mechanism placement: We implemented connection redirection as part of the Linux kernel assuming the following deployment models: (1) In the case the kernel patch goes upstream, newer kernel version will support it (2) If not, cloud providers can offer VM images with the modified kernel which cloud tenants can leverage to benefit from CRAB. However, these assumptions are not fundamental to CRAB. We now discuss how CRAB's advantages can be retained with alternative placements of connection redirection that the client and server kernels remain agnostic to.

Cloud providers implement engines either in software [240, 75, 115], or in hardware [116] that accelerate their virtual networking infrastructure. These engines apply address translation rules and encapsulate and decapsulate packets. Connection redirection can be supported by those engines, instead of the guest kernels. On the client side receiving a `SYN-ACK` with the `Connection Redirect` option will create two new rules that will perform Source Network Address Translation (SNAT) for the received packets and Destination Network Address Translation (DNAT) for transmitted packets respectively. The engine will overwrite the DIP with the VIP in the received option for incoming packets, and vice-versa for the transmitted packets. The server-side implementation will create a short-lived rule that on receiving a `SYN` packet with the redirection option to echoes the option in the outgoing `SYN-ACK`. The downside of such an implementation is that it involves packet modifications on the critical path that can incur performance overheads in a software-based stack. Despite the similarities with the agent-based load balancing in Section 7.2, supporting CRAB on the host infrastructure still enables guests to benefit from the centralized load balancing policies and easy and fast updates to the server pool.

Alternative Transports: While we focus only on TCP, here, we discuss how the core ideas behind CRAB apply to other connection-oriented transport protocols, in particular QUIC. QUIC [211] is a low-latency transport protocol designed originally for HTTPS traffic.

While QUIC runs over UDP, it still retains the notion of a connection that is established between a client and a server after a handshake. QUIC also allows a 0-RTT connection establishment

for endpoints that have already communicated in the past. After the initial handshake, the connection is associated with a ConnectionID that defines the connection. Load balancers use this ConnectionID to forward packets from the same connection to the correct back-send server [211]. ConnectionIDs also enable seamless connectivity during endpoint migrations (address changes).

We believe CRAB naturally extends to QUIC. Our proposed connection redirect TCP option is, in fact, a subset of QUIC's connection migration functionality. In a CRAB-enabled QUIC, the 1-RTT handshake has to be performed over the load balancer, since no migration is allowed during connection establishment. Then the server initiates a migration to its DIP instead of the VIP by leveraging the ConnectionID. Following 0-RTT connection establishments can use the DIP directly, thus bypassing the load balancer.

Caveats: Unlike L7 load balancing and DNS-based load balancing that can be deployed by both cloud providers and cloud tenants, L4 load balancing and CRAB require IP spoofing, namely the ability to send an IP packet with a source IP that is different from the local IP. CRAB relies on the load balancer sending packets to the back-end servers with the client IP as source IP. However, IP spoofing is not available for cloud tenants. Thus, CRAB can only be deployed by cloud providers, substituting or complementing their existing L4 load balancing offerings.

CRAB and our proposed connection redirect feature can affect existing mechanisms that perform connection tracking. Such mechanisms can be either implemented in software, such as `conntrack` [342] and Receive Flow Steering (RFS) [351], or in hardware, such as Intel's Application Targeting Routing (ATR) [345]. The goal of such mechanisms is either performance optimization (*e.g.,* interrupt steering for locality) or monitoring (*e.g.,* `conntrack-tools`). In our current implementation, we do not handle such potential violations. In our unloaded latency experiments, we steered all NIC interrupts to the core running the client application while in our throughput experiments all cores were constantly busy. Thus, we did not observe any performance degradation due to hindering of connection tracking mechanisms and specifically ATR.

CRAB for Kubernetes NodePort: We only showed the benefits of using connection redirection in load balancing with CRAB when using the central CRAB middlebox. However, connection redirection can have other use cases in a cloud setup, too. One such use-case arises from the Kubernetes ecosystem and the use of NodePort [346].

The NodePort configuration exposes a service running on a Kubernetes cluster on every cluster node independent of whether pods are running this service on the specific node. This way a service can be placed behind an L4 load balancer having all cluster nodes in its server pool. In cases when a node receives traffic for a service that does not run locally, it forwards this traffic to a node that does using kube-proxy. A downside of this approach is that all traffic flows through this intermediate hop, deteriorating the client's perceived latency. To deal with the increased

latency, administrators can configure Kubernetes with `externalTrafficPolicy=Local`, so that new connections get rejected if there are no local service instances. This, however, is only a remedy rather than a solution.

Connection redirect can reduce the latency overhead in the above scenario as follows: The initial server that is assigned the load balancer but does not run the service can add the redirect TCP option before forwarding the SYN packet to the node running the service. That node will then echo back the option as usual, but instead of the client, this option will be intercepted by the stateful load balancer which will then update the target for future packets from this specific connection.

7.7 Related work

Load balancing is performed at different layers of the networking stack. Below L4 load balancers and completely orthogonal and complementary to our work is network load balancing. Hash-based approaches such as ECMP [330, 320], or more complex approaches such as Conga [6] and alternatives [130, 150, 190, 354, 380] make sure that the multiple paths inside 3-tier datacenter clos topology are equally loaded, thus reducing queuing at the datacenter switches.

CRAB can be thought of as a lightweight L4 load balancer that bypasses the typical L4 load balancing limitations. There are numerous load balancer implementations in software [275, 11, 96, 189, 286], and hardware [27, 251, 124, 292, 242] with stateful or stateless designs. Stateful designs suffer from scalability limits, while stateless designs suffer from suboptimal load balancing policies. To overcome the space limitations in stateful load balancers Kablan et al. [177] suggested using external stores for the load balancing state. CRAB achieves the best of both worlds since it can implement complex load balancing policies and avoid PCC violations while remaining stateless.

The works most closely related to ours, though, are the following. Duchene et al. [92] target a specific to Multi-path TCP (MPTCP) [118] problem and ensure that the different TCP connections within a single MPTCP connection are routed to the same server, using a similar mechanism to connection redirect, in which the back-end server advertises its IP to the client during connection setup. R2P2 [202] similarly to CRAB enables load balancer bypass both on the transmit and receive path from the client perspective, but does so by employing a novel transport layer that exposes individual RPCs. Cheetah [27] exposes an identifier to the clients that is used by the load balancer for forwarding, thus achieving policies equivalent to a stateful load balancer without keeping state at the load balancer. However, cheetah's load balancer is always on the critical path, unlike CRAB. Finally, QUIC's connection migration feature [350] serves as a subset of the proposed connection redirect option.

The above solutions can only be deployed by cloud providers and offered as services to the cloud tenants. Tenants that want to have more control over their infrastructure and how load

balancing is performed can deploy L7 or DNS-based load balancing. Examples of open-source software that provides such services is NGINX [269], varnish [355], haproxy [146], envoy [99] *etc.* for L7 and bind [341] and dnsmasq [343] for DNS. L7 load balancers can implement more elaborate load balancing policies than CRAB since they can also do request-level load balancing but incur more significant costs since they run in userspace. DNS-based load balancing can display similar performance to CRAB but suffers from the problems associated with caching.

Cloud providers use cluster schedulers such as Borg [356], Kubernetes [205, 47], Mesos [152], Docker Swarm [328] to provision virtual resources for container or VM workloads aiming to maximize utilization without violating customer SLOs. CRAB can be used as part of the above solutions helping their elasticity. CRAB dramatically reduces the resources necessary for load balancing, thus leaving more available resources for the above schedulers to run client workloads on.

7.8 Chapter Summary

CRAB is a novel design for internal load balancers in the public cloud. It depends on a new TCP option that enables connection redirection; thus, the load balancer participates only in the connection establishment. Unlike traditional L4 load balancers, CRAB does not impact the latency or bandwidth of established connections. CRAB can support the same rich load balancing policies as traditional load balancers, but without keeping per connection state within the load balancer. CRAB is backward compatible with TCP, and easily deployable on the public cloud with minor kernel modifications.

8 SLO-Aware TCP Flow Control

8.1 Introduction

The "cloud era" is built on top of at least two nearly ubiquitous paradigms: TCP/IP and remote procedure calls (RPC) [38]. Both paradigms are used to connect to mega-datacenters (*e.g.,* `https`) as well as within datacenters, where they connect multiple tiers of servers with wide fan-in/fan-out flow patterns and strict tail-latency service level objectives (SLO) [31, 80]. TCP has emerged as the main transport protocol on top of commodity Ethernet for latency-critical RPCs. The reliable, ordered byte stream provided by TCP serves as the basis for other higher-level, application abstractions and guarantees, such as exactly-once RPC semantics. Non-commodity alternatives, *e.g.,* RDMA over Infiniband and RDMA over Lossless Ethernet (RoCE) have specific hardware requirements or expose alternative APIs, that limits their wide adoption, despite potentially reduced round-trip times (RTT).

A datacenter, though, differs radically from the assumptions considered during the initial TCP design [66]. Specifically, modern datacenters are uniformly designed and built upon low-latency Ethernet fabrics in Clos topologies. Those fabrics comprise of commodity cut-through switches with shallow buffers that have a few hundreds of nanoseconds of switching latency. Such a design guarantees unloaded RTTs in the scale of µs and a few Pbps of bisection bandwidth [320]. Despite TCP's extensive use within a datacenter, most of its mechanisms remain as originally designed, such as the sliding window. TCP's sliding window is managed by both the congestion and flow control logic. We make the distinction between congestion control algorithms focusing on in-network congestion, while flow control focusing on queueing on the end-hosts. Improvements throughout the networking stack, though, have mostly focused on the in-network congestion. Approaches such as [7, 8, 6] reduced congestion and buffer utilisation, nearly eliminated packet drops, improved fabric utilisation, and reduced latency jitter.

In contrast, the endpoints have received less consideration. In particular, the core flow-control mechanism of TCP is used today primarily to prevent packet loss at the end-hosts, but without any particular consideration for end-to-end latency. The initial purpose of flow control is

to avoid overwhelming the receiver, which can lead to packet drop. This was a significant concern during the initial design of TCP due to the limited amount of DRAM in computers of a few decades ago. Modern servers nowadays have abundant DRAM, and can potentially accommodate a large number of packets. Applications with strict tail-latency SLOs, though, require minimal queueing on the server side. For a datacenter RPC server that handles µs-level requests from multiple sockets (fan-in), the queuing on the server side will account for a significant proportion of the client end-to-end latency, given that the service time and the RTT combined account for a few µs. Latency-critical RPCs, such as key-value stores, usually depend on short requests and responses of a few bytes [14]. So, an in-memory, latency-sensitive RPC server will violate a strict SLO after queuing only a few dozens of short messages, which collectively require only a tiny fraction of the server's memory.

This chapter suggests to revisit the notion of flow-control specifically for latency-sensitive, µs-scale, kernel-bypassed RPC services on top of TCP. According to our proposal, the destination signals its availability back to the source in an application-independent manner, as with TCP's standard flow control. Unlike the standard approach, though, the signal is based on the expected wait and service times of the server and takes into consideration the specific application-level tail-latency SLOs (rather than available memory).

Our flow-control mechanism uses a token bucket per connection to control the incoming request rate and maintain it at a level so that the application SLOs are not violated. The clients self-pace their requests based on available tokens. The server controls the size and the fill-rate of the bucket based on (a) the trailing estimate of the service time distribution of RPCs, (b) the SLO, and (c) the number of connections.

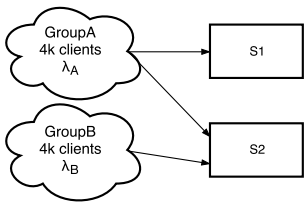


Figure 8.1 – Replicated RPC experiment setup

We implement a proof-of-concept TCP stack with a latency-aware flow control mechanism by simply re-purposing the TCP sliding window. The new mechanism requires no changes to the TCP header format or additional messages. Our implementation on top of Intel's DPDK [88] is suitable to evaluate the effectiveness of the approach for µs-scale tasks. Our evaluation of the mechanism in a series of synthetic microbenchmarks with different service time distributions shows that it can accurately identify the load that will violate the latency SLO and maintain the throughput at that level. Moreover, we show the benefits of using this mechanism in cases of replicated services in order to avoid overloaded servers.

8.2 Motivation and Background

In a latency-aware flow control mechanism, the two ends, client and server, communicate, so that the client adjusts their offered load after the server's directives, based on how the server

performs and the agreed SLO. In a scenario with replicated RPC-servers, this mechanism can be used to dynamically adjust the incoming load to the servers, so that the application tail-latency SLOs are not violated. Figure 8.1 provides a motivating example that applies the end-to-end argument for replicated RPC services: 8K clients are split into two groups, where the first group can interchangeably select between server 1 (*S1*) and server 2 (*S2*), but the second group (for some reason) can only use *S2*. The relative arrival rate of requests from the two groups is unknown and varying, and the service time to process each request is also unknown. By incorporating tail-latency awareness into flow control, each client should be able to independently determine when and where to send requests so that the SLO is never violated.

8.2.1 Existing TCP Flow Control

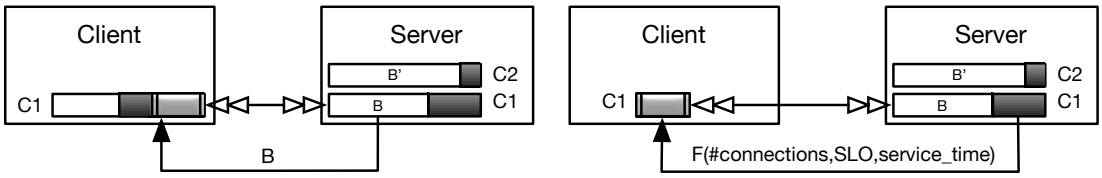
TCP, as a connection-oriented transport protocol, implements flow control per connection. The flow control mechanism depends on the size of the available receive socket buffer communicated between the two endpoints with every packet exchanged. The sender uses this information in conjunction with the current congestion window size to decide how many bytes to send, and sends the maximum amount allowed.

From an implementation perspective, flow control is implemented via a 16-bit header field, with window-scale option exchanged during the 3-way handshake [165]. On each endpoint the size of the receive buffer can be configured per connection during runtime through a `setsockopt()` call. However, it can not be configured to less than 4kB. A server with a high client count (high fan-in) must therefore accept potentially multiple RPCs from each client, before the existing flow control starts throttling each client independently.

Figure 8.2a summarises the existing TCP flow control mechanism. It shows a connection between a server and a client. The server also has second connection to another client. There are buffers on both ends of the connection that are partially occupied. TCP's existing flow control mechanism signals back the available per connection buffer space (B) as part of the packets exchanged.

8.2.2 TCP Flow Control in KBNets

Kernel-bypass networking introduces additional issues related to the implied semantics of flow-control. TCP's flow control accounts for the bytes processed by the TCP stack but *not* processed by the application. Thus, it assumes certain asynchrony between network and application processing. This assumption is invalid for kernel-bypass implementations with a symmetric, run-to-completion design inspired by middlebox dataplanes[34, 290]. Because of the tight coupling of network and application processing, there are no buffered data between the two stages. Thus, the semantics of TCP's flow control become vague. It appears as if receiver buffers are never filled up, despite incoming packets being queued or dropped before



(a) TCP Flow Control (b) Latency-Aware Flow Control

Figure 8.2 – One-way Flow Control FeedBack Loop. C_i indicates a connection, dark grey indicates occupied buffer space, light grey indicates the sliding window, B is the available receive buffer space.

TCP and application processing. The alternative approach dedicates threads to network processing, other threads to application processing, and the two groups communicate over interprocess communication, *e.g.,* shared memory [167]. Here, the TCP flow control semantics remain unchanged.

8.3 Design

Our goal is to reduce and control buffering on the path of an RPC on top of TCP. To do so, we design a latency-aware flow control mechanism for TCP, specifically targeting latency-critical RPC services.

Figure 8.2b summarises the proposed design. On the RPC-client side, buffering is limited to the sliding window, which ensures that requests can be sent without delay to the server. On the server side, we change the semantics of the flow control signal to take into consideration the total amount of connections on the server, and the application SLOs and service time. The flow-control logic on the server should be able to predict the overall incoming load that will violate the latency SLO and maintain throughput below that level across all connections. We set the following design requirements: (a) There should not be extra messages specifically for flow control. (b) The TCP header format should remain intact. (c) The mechanism should be agnostic to the service time distribution.

Predicting the SLO-violating load: Since we want to leverage flow control, a throughput-oriented mechanism, to control the end-to-end request latency, we need to understand and approximate the correlation between the incoming rate of requests and the end-to-end latency of each request. To do so, we use some basic queueing theory. Although TCP is a connection-oriented transport protocol, we focus on RPC services and abstract the basic system functionality. Thus, a system can be described by its service time distribution, the incoming distribution of requests, and the number of workers and queues. We use Kendall's notation to describe queuing models, where in the following expression `A/S/n`, `A` is the inter-arrival distribution, `S` is the service time distribution, and `n` is the number of workers. The scheduling policy implied is first-come-first-served (FCFS). The end-to-end request latency

is the sum of the propagation delay, the wait time in the queue, and the service time. The propagation delay depends on the request size and the network characteristics. The service time distribution depends on the application. Thus, we focus on how to control the wait time and provide wait time SLOs (WSLOs), namely an upper limit on the time each request might wait to be served.

According to queuing theory, the average wait time in an `M/G/1` system can be expressed in a closed form that depends on the system load, and the average and standard deviation of the service time. This implies that in any system with a Poisson arrival distribution and a single worker, we can control the average wait time by controlling the system load, independently of the service time distribution. Equation 8.1 expresses the average wait time (T_w), as a function of the system load (ρ), and the service time standard deviation (σ_{T_s}) and average (T_s) [214].

$$T_w = 1/2 \frac{\rho T_s}{1-\rho}[1 + (\frac{\sigma_{T_s}}{T_s})^2] \tag{8.1}$$

SLOs, though, are expressed at some percentile. Since there is no closed form that expresses the wait time percentiles, we will use the central limit theorem for heavy traffic queueing systems [194] to approximate them. According to that, in any G/G/N queuing system under heavy traffic load, the wait time distribution could be approximated by an exponential distribution. Based on that we can approximate any percentile of the wait time distribution, since we know the distribution to be exponential and its average from Equation 8.1. Vice versa, we can set an upper limit in the wait time for a certain percentile, and predict the system load that will violate this WSLO. To do so, first we need to estimate the average wait time Tw_{target} at the SLO. Equation 8.2 computes the average of an exponential distribution whose value at the p-percentile is Tw_{slo}. β is the scale parameter of the exponential distribution. If we substitute Tw with Tw_{target} in Equation 8.1, and solve for ρ, we get the load that will violate the WSLO. For example, for a fixed service time distribution ($\sigma_{T_s} = 0$) with $Ts = 10$, and a 99-th percentile WSLO of 100($Tw_{slo} = 100$ and $p = 0.99$), we predict that the WSLO will be violated when $\rho = 0.81$

$$\begin{aligned} Pr[X \leq Tw_{slo}] &= p \\ 1 - e^{-\frac{Tw_{slo}}{\beta}} &= p \\ Tw_{target} = \beta &= -\frac{Tw_{slo}}{\ln(1-p)} \end{aligned} \tag{8.2}$$

Flow-control Mechanism: We incorporate the above result in the existing TCP's flow control infrastructure. The operator should only define the WSLO as the amount of wait time allowed

at a certain percentile and the system should adjust accordingly. Although TCP connections are symmetric, in the following explanation we focus on the case of an RPC server (receiver) that needs to apply flow control to the incoming requests from different clients (senders).

We first describe the functionality of the receiver running a single RPC service. For simplicity, we consider a single-threaded RCP server and we analyse the multi-threaded case at the end of the section. The receiver calculates the maximum allowed incoming rate according to the WSLO, based on the formulas in Section 8.3, and prevents the clients from sending faster than this rate. To estimate the average and standard deviation of the service time, every receiver maintains a global moving average of the application service time per request. Moreover, to avoid unnecessary client throttling, every receiver maintains a moving average of the request inter-arrival time across all connections, and throttles only when the overall load is close to the maximum allowed one.

The throttling mechanism depends on a per-connection token bucket algorithm controlled by the receiver. Every token corresponds to a request. If the incoming rate is lower than a certain percentage of the maximum allowed load, the receiver allocates a fixed number of tokens per connection. For our experiments we set the threshold at 80% of the maximum allowed load. After that threshold throttling is necessary. The receiver distributes the maximum allowed load across all connections. Load distribution can consider any connection priority scheme depending on the application logic. The receiver translates the per connection load to a number of tokens per connection that periodically replenishes, and communicates the amount of available tokens to the receiver through the TCP header. If there are no tokens available, the sender should remain silent till the new replenishment. To do so without the need of extra messages, the receiver notifies the sender about the duration it should remain silent, again through the TCP header.

Unlike the existing TCP semantics that depend on the socket API with intermediate buffering on the send path, we assume a zero-sized send buffer other than the sliding window. If there is a request to be sent, the sender immediately sends it as long as there are available tokens (one token per request). If not the send fails. Thus, the application can decide whether it should drop the request, buffer it in the application space and wait, or try using another connection, if any. Compared to the existing TCP implementation, sends will fail more often. A failed send, though, now implies that if this request was actually send, it would probably be an SLO violation. The sender can send again either after a token replenishment, or at the end of the idle period as defined by the receiver.

A multi-threaded server would operate on a similar rational. We assume that every thread serves incoming requests independently from the others, and there is a static mapping of connections to threads. Every thread runs a run-to-completion loop. This is a popular design adopted by latency critical applications, such as NGINX [266] and memcached [250], and operating systems, such as IX [34]. In this design, every thread maintains its own moving averages for service time and inter-arrival distribution that correspond to the connections it is

responsible for, and implements flow control based on the load it individually receives.

Assumptions and Limitations: The use of closed forms in the above design depends on the two following assumptions: (1) the RPC inter-arrival distribution is exponential (Poisson process); (2) the system is modelled as $M/G/1$, namely each core operates as a standalone entity and there is no connection sharing across cores. If either of the two assumptions is violated, we can still use heuristics to correlate the incoming load with the queueing delay as part of a training phase, and then use the same mechanism to keep the incoming load below the threshold that emerged after the training phase.

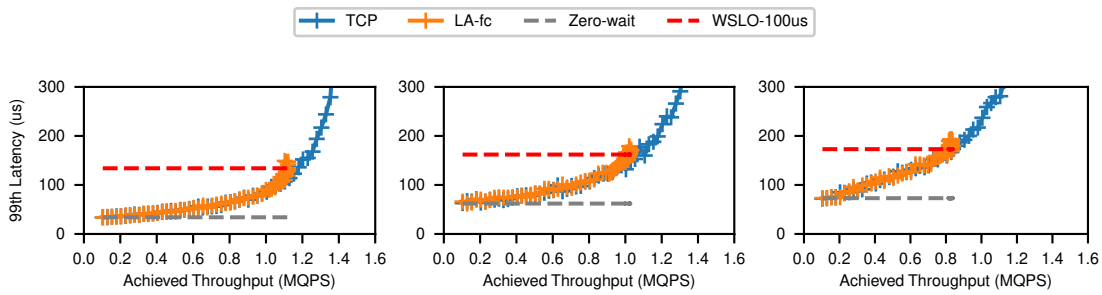
Finally, note that our latency-aware flow control logic is integrated within TCP without using any extra packets, but is in fact independent of the actual transport used. For example, it could trivially be implemented on top of RDMA by using either extra control messages from the server back to the clients, or by having the clients read the load from the server using one-sided remote read operations.

8.4 Implementation

We implemented the above design in a userspace TCP/IP stack and used it in a sample RPC client and server that are built on top of Intel's DPDK [88]. The TCP/IP stack does not make any assumption about the kernel-bypass paradigm, namely, how to split network and application processing across threads, and can be used both in a symmetric and an asymmetric setup. We built the client and server applications after the symmetric paradigm, where every thread runs both network and application processing.

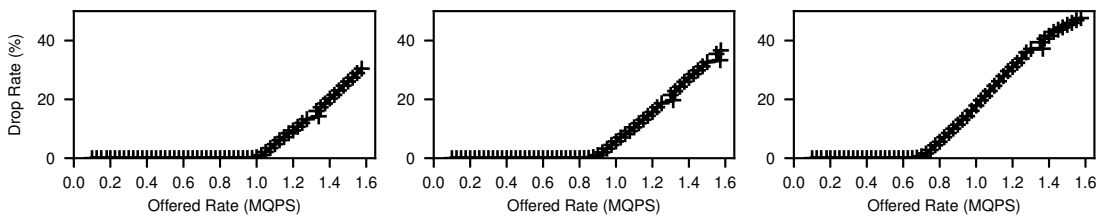
Since we specifically target latency-critical RPC services, we didn't expose a POSIX-like API for our proof of concept TCP stack. Instead, we implemented an event-based API similar to IX [34]. In this API, the application has direct access to the packets received. Whenever there is an incoming packet, the application gets notified and receives a pointer and the length of the received payload. Similarly, whenever the application wants to send a packet, writes directly to the `mbuf` to avoid extra copies. Before sending, every sender explicitly asks for a new `mbuf` to put the data to be sent at the right offset. Finally, we implemented two extra function calls to collect the per RPC application processing time. The application calls these two functions at the beginning and end of processing each individual RPC, equivalently.

We implemented the proposed flow-control mechanism by leveraging the existing 16-bit receive window without introducing any extra messages or changing the TCP header. Unlike the existing semantics though, where this field carries a number of bytes, we need to overload it with dual semantics, since it can represent either a number of tokens or the duration of idle time. We use the most significant of the 16 bits as a mode switch. If not set, the lower 15 bits encode the number of available tokens. If set, they carry the number of μs for which this particular connection should remain idle. By configuring the token replenishment



(a) Fixed ($\bar{S} = 10\mu s$) (b) Exponential ($\bar{S} = 10\mu s$) (c) Bimodal ($\bar{S} = 10\mu s$)

Figure 8.3 – 99th percentile tail latency versus throughput for the three service time distributions with 10µs mean service time and the two flow control mechanisms, TCP and the proposed latency-aware (LA-fc).



(a) Fixed (b) Exponential (c) Bimodal

Figure 8.4 – Drop rate as a function of the offered throughput in the case of LA-fc for the 3 service time distributions.

rate equivalently, 15 bits are enough to carry the available tokens per connection. In our experiments we replenished each connection's tokens every 500 µs. For the idle time, we assume that no connection should remain idle for more than 32 ms, which is approximately the maximum duration in µs encoded in 15 bits. If for any reason, the 15 bits are not enough, we can employ the existing window scaling mechanism.

We didn't implement any congestion control in our proof-of-concept TCP stack, since our experiments are application-throttled. Existing window-based congestion control schemes are not effective for this particular type of workload consisting of very small messages [14] that fit in a single packet, across a large number of connections, in an environment with µs RTTs. Congestion control for such workloads is an open research problem [59, 252, 383, 43] with recent research proposals [263] suggesting a generic congestion control agent that can be used in kernel-bypass networking stacks. Such an approach is a good fit for our networking stack, but we leave this for future work.

8.5 Evaluation

To evaluate our design we implemented a synthetic RPC service based on our experimental TCP stack. We use fixed-size requests and responses of 8 bytes. Each request encodes the amount of processing time required by the server. Once the server receives the request, spins for the amount of time specified and echoes the received value back. This way we can emulate any service time distribution and evaluate whether our system can adjust accordingly.

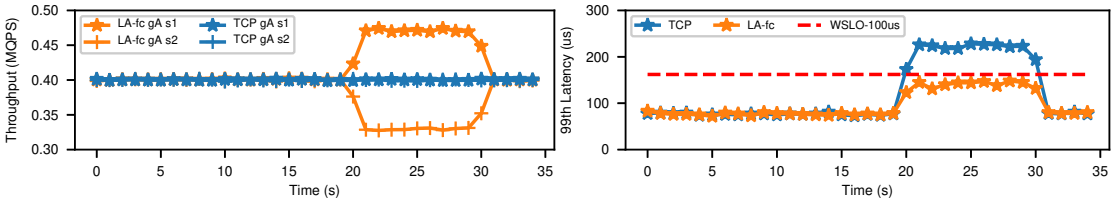
Our experimental setup consists of a cluster of 4 client and 2 server machines connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The client machines are Xeon E5-2637 @ 3.5 GHz with 8 cores (16 hyperthreads), and the server machines are Xeon E5-2650 @ 2.6 GHz with 16 cores (32 hyperthreads). All machines are configured with Intel x520 10GbE NICs (82599EB chipset). We connect the clients and the server to the switch through a single NIC port each. On both clients and servers we use only the one NUMA node.

We run two types of experiments. In the first type we emulate multiple independent clients having a single connection to the RPC server and we show how the system can identify the load that will violate WSLO, and maintain the incoming request rate lower than that level. In the second type of experiment, we show how this mechanism can be used in a replicated RPC service, so that clients avoid overloaded replicas, and thus achieving better tail-latency.

Single-node RPC service: In this experiment we use 4 client machines and 1 server machine. Each client machine opens 1024 connections spread across 8 threads. Every client thread generates requests with a Poisson arrival distribution, randomly chooses one of the available connections, and sends the request to the server. This workload is equivalent to 4096 independent clients generating requests with a Poisson arrival against the RPC server. If flow control allows it, the client sends the request to the server. Otherwise, the request is dropped. Note, that this is an extreme case that will help us evaluate the maximum throughput loss because of the new flow control mechanism. In a more realistic scenario, the client could decide to wait for a certain amount of time for the connection to become available before dropping the request depending on the application SLOs.

We use three different service time distributions with the same average of 10µs, but different dispersion. We consider a fixed, an exponential and a bimodal distribution where 90% of the requests require 5 µs of processing time and the rest 10% requires 55 µs. We run the client and the server both with the standard TCP flow control mechanism and with the proposed latency-aware flow control mechanism, and we set a WSLO of 100µs at the 99-th percentile.

Figure 8.3 shows the 99-th percentile end-to-end latency in µs measured at the client as a function of the achieved throughput in million requests per seconds for the two flow control mechanisms, TCP and latency-aware. The server runs on 16 cores, so with an average service time of 10µs, the theoretically maximum possible throughput is 1.6 million requests per second. We limit the y-axis at 300 µs. The red dashed line is the WSLO defined as 100 µs above



(a) Throughput over time

(b) Latency over time

Figure 8.5 – GroupA (gA) throughput against server1 (s1) and server2 (s2), and 99th percentile tail latency for the two flow control mchanisms, TCP and the proposed latency-aware (LA-fc)

the unloaded latency. Firstly, we observe that in the case of TCP flow control, latency can increase beyond the SLO. TCP is agnostic to the application SLO and and there are multiple clients with outstanding requests. In the case of latency-aware flow control, the system manages to throttle the incoming rate according to the WSLO. Moreover, we can see that the maximum allowed load changes according to the service time distribution. This load increases as the service time dispersion decreases.

In this experiment we traded the drop rate for better tail-latency, since clients instantly drop their request if the connection is not available. Figure 8.4 shows the drop rate as a function of the offered request rate for the same three distributions of service time. In this plot, it becomes obvious that our mechanism can decide when to start throttling based on the different service times. Clients start dropping requests at different request rates across the three service time distributions. As expected, as the service time dispersion increases, client start dropping requests earlier.

Multi-node RPC service: This experiment addresses the motivating example of Figure 8.1 by showing how the mechanism can be used by clients of a replicated RPC service to avoid the overloaded replicas. We assume a replicated RPC service across two servers where the two servers, *S1* and *S1*, can be used interchangeably. We split the client machines into two groups. GroupA emulates 4096 independent clients that have one connection to each server and generate requests with a Poisson arrival (λ_A). According to the client logic, each client randomly chooses between the two connections. If the chosen connection can be immediately used for sending, they send their request. Otherwise they try their other connection. If this one is available, they use this to send the request, else they drop it. GroupA generates a constant load of 800 kQPS with a 10-µs exponential service time against the two servers. The second group of clients, groupB, again consists of 4096 independent clients but they can only use *S2*. GroupB clients generate a constant load of 800 kQPS with a 10-µs exponential service time starting at $t_0 = 20$ for 10 seconds and then they stop. We measure and plot the throughput and latency achieved by groupA for the two flow control mechanisms, TCP and latency-aware.

Figure 8.5a shows the achieved throughput for groupA, and how it is distributed across the two servers. In the case of TCP, groupA selects both *S1* and *S2* with the same likelihood,

independently of groupB's behaviour. However, in the case of the latency-aware flow control mechanism, groupA starts showing preference to *S1* at $t_0 = 20$. After groupB starts loading, *S2* operates beyond its WLSO, so it starts throttling. As a result groupA's connection to *S2* can not support the same load as before. Thus, *S1* is chosen more often. GroupA's request rate to *S1* increases, and the rate to *S2* decreases, while the aggregate throughput remains the same. The drop rate for groupA is close to zero. Regarding groupB throughput, although not shown in the graph, it is 800 kQPS in the case of TCP, while it drops to around 660 kQPS in the case of latency-aware flow control. This is in accordance with the results in Figure 8.3b for *S2*, since 1 MQPS (330 kQPS from groupA and 660 kQPS from groupB)is approximately the maximum allowed load for the exponential distribution with a WSLO of 100 µs. We could avoid dropping groupB's throughput by choosing a different load allocation policy in *S2* that favours groupB connections.

Figure 8.5b shows the end-to-end tail-latency as it is measured by the groupA clients, by choosing either *S1* or *S2* according to the client logic. The red dashed line is the WSLO for the exponential distribution as shown in Figure 8.3b. As expected, groupA tail-latency increases when groupB starts loading server2. In the case of TCP, though, the WSLO is violated since *S2* is still selected with the same likelihood. As a result, *S2* serves approximately 1.2 MQPS (800 kQPS from groupB and 400 kQPS from groupA) resulting in a 99-th percentile latency around 230 µs. This result is in accordance with the TCP result in Figure 8.3b. The latency-aware flow control mechanism manages to adjust the *S2* throughput so that the WSLO is not violated, given that groupA's 99-th percentile latency is kept below 160 µs.

8.6 Related Work

Several kernel-bypass systems implement TCP/IP stacks [167, 34, 94, 290, 239, 273, 277, 316, 89, 227], and remain fully compatible with the exact TCP semantics. Similarly to TCP, protocols such as QUIC [211] and HTTP2 [35], also, implement flow control mechanisms based on buffer occupancy, and remain agnostic to latency SLOs.

Token bucket algorithms are the canonical way of implementing flow control in any kind of communication in fields such as wide-area networks [338], networks-on-chip [168], storage [332, 368, 382, 196], and network congestion control [59].

There are several research proposals on changing the semantics or implementation of TCP, according to application needs, such as sharing the congestion window [161] for faster convergence, advertising the send buffer occupancy for better load balancing and network utilization [5], or adaptively changing the send buffer size for streaming [133].

Finally, there a several research proposals dealing with selecting servers in a replicated service and reducing tail-latency because of strugglers [80, 326, 145, 253, 369, 10]

8.7 Chapter Summary

We presented a latency-aware flow control mechanism specifically for latency-critical RPCs that use TCP. We introduced the notion of wait time SLO and we showed how our flow control mechanism prevents applications from violating their WSLOs. Finally, we showed how this mechanism can benefit clients of replicated services to avoid overloaded replicas, and thus reduce the end-to-end tail-latency.

New Abstractions for RPCs Part III

9 Request Response Pair Protocol (R2P2)

9.1 Introduction

Web-scale online data-intensive applications such as search, e-commerce, and social applications rely on the scale-out architectures of modern, warehouse-scale datacenters to meet service-level objectives (SLO) [29, 80]. In such deployments, a single application can comprise hundreds of software components, deployed on thousands of servers organized in multiple tiers and connected by commodity Ethernet switches. The typical pattern for web-scale applications distributes the critical data (*e.g.,* the social graph) in the memory of hundreds of data services, such as memory-resident transactional databases [337, 325, 358, 122, 361], NoSQL databases [304, 254], key-value stores [90, 229, 250, 376, 271], or specialized graph stores [45]. Consequently, online data-intensive (OLDI) applications are deployed as 2-tier applications with root servers handling end-user queries and leaf servers holding replicated, sharded data [244, 30]. This leads to a high fan-in, high fan-out connection graph between the tiers of an application that internally communicates using RPCs [38]. Each client must (a) fan-out an RPC to the different shards and (b) within each shard, choose a server from among the replica set. Moreover, each individual task can require from only few microseconds (µs) of user-level execution time for simple key-value requests [229] to a handful of milliseconds for search applications [148].

To communicate between the tiers, applications most commonly layer RPCs on top of TCP, either through RPC frameworks (*e.g.,* gRPC [139] and Thrift [333]) or through application-specific protocols (*e.g.,* Memcached [250]). This leads to a mismatch between TCP, which is a byte-oriented, streaming transport protocol, and message-oriented RPCs. This mismatch introduces several challenges regarding RPC load balancing, RPC flow control, *etc.*.

We propose a new transport protocol for datacenter RPCS called *Request-Response Pair Protocol (R2P2)*. R2P2 exposes RPCs as first-class citizens of the datacenter not only at the client and server endpoints, but also in the network. Endpoints have direct control over RPC semantics, do not suffer from head-of-line blocking because of connection multiplexing, and can limit buffering at the endpoints. The design also enables RPC-level processing capabili-

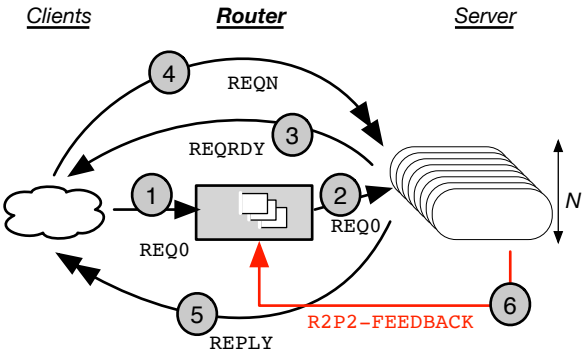


Figure 9.1 – The R2P2 protocol for a request-reply exchange. Each message is carried within a UDP packet. Single arrows represent a single packet whereas double arrows represent a stream of datagrams.

ties for in-network software or hardware middleboxes, including scheduling, load-balancing, straggler-mitigation, consensus and in-network aggregation. R2P2 breaks the point-to-point RPC communication assumptions, and separates RPC policy enforcement from message streaming.

In this chapter we describe the design and implementation of the R2P2 protocol, while in the next chapters(Chapter 10, Chapter 11, Chapter 12) we see how we can use the new protocol to implement different RPC policies in the network. The R2P2 source code is available at `https://github.com/epfl-dcsl/r2p2`.

9.2 R2P2: A transport protocol for RPCs

We propose R2P2 (*Request-Response Pair Protocol*), a UDP-based transport protocol specifically targeting latency-critical RPCs within a distributed infrastructure, *i.e.,* a datacenter. R2P2 exposes the RPC abstraction to the network, thus allowing for efficient in-network request-level load balancing.

R2P2 is a connectionless transport protocol capable of supporting higher-level protocols such as HTTP without protocol-level modifications. Unlike traditional multiplexing of the RPC onto a reliable byte-oriented connection, R2P2 is an inherently request/reply-oriented protocol that maintains no state across requests. The R2P2 request-response pair is initiated by the client and is uniquely identified by a triplet of $< src_IP, src_port, req_id >$. This design choice decouples the request destination (set by the client) from the actual server that will reply, thus breaking the point-to-point RPC communication semantics and enabling the implementation of any request-level load balancing policy.

Figure 9.1 describes the interactions and the packets exchanged in sending and receiving an RPC within a distributed infrastructure that uses a request router to load balance requests

across the servers. We illustrate the general case of a multi-packet request and a multi-packet response.

1. A `REQ0` message opens the RPC interaction, uniquely defined by the combination of source IP, UDP port, and an RPC sequence number. The datagram may contain the beginning of the RPC request itself.
2. The router identifies a suitable target server and directs the message to it. If there is no available server, requests can temporarily queue up in the router.
3. If the RPC request exceeds the size of data in the `REQ0` payload, then the server uses a `REQready` message to inform the client that it has been selected and that it will process the request.
4. Following (3), the client directly sends the remainder of the request as `REQn` messages.
5. The server replies directly to the client with a stream of `REPLY` messages.
6. The servers send `R2P2-FEEDBACK` messages to the router to signal idleness, availability, or health, depending on the load balancing policies.

We note a few obvious consequences and benefits of the design: (i) Given that an RPC is identified by the triplet, responses can arrive from a different machine than the original destination. Responses are sent directly to the client, bypassing the router; (ii) there is no head-of-line blocking resulting from multiplexing RPCs on a socket, since there are no sockets and each request-response pair is treated independently; (iii) there are no ordering guarantees across RPCs; (iv) the protocol is suited for both short and long RPCs. By avoiding the router for `REQn` message and replies, the router capacity is only limited by its hardware packet processing rate, not by the overall amount of size of the messages.

Unlike protocols that blindly provide reliable message delivery, R2P2 exposes failures and delays to the application. R2P2 follows the end-to-end argument in systems design [311]. A client application initiates a request-response pair and determines the failure policy of each RPC according to its specific needs and SLOs. By propagating failures to the application, the developer is free to choose between *at-least-once* and *at-most-once* semantics by re-issuing the same request that failed. Unlike TCP, failures affect only the RPC in question, not other requests. This is useful in cases with fan-out replicated requests, where R2P2 can provide system support for the implementation of tail-mitigation techniques, such as hedged requests [80].

While novel in the context of µs-scale, in-memory computing, the connection "pair" is similar in spirit to the "exchange" that is the core of the SCSI/Fibre Channel protocol (FCP [111]). For example, a single-packet-request-multi-packet-response RPC over R2P2 would be similar to SCSI `read` within a single fibre channel exchange. Equivalently, an R2P2 multi-packet-request-single-packet-response would be similar to a SCSI `write`.

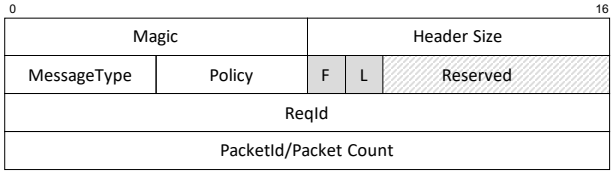


Figure 9.2 – R2P2 Header Format

Message	Description
REQUEST	A message carrying an RPC request
REPLY	A message carrying an RPC reply
REQRDY	Sent by the server to the client to ack the REQ0 of a multi-packet request
R2P2-FEEDBACK	Sent by the server to the router
DROP	Sent by the router or the server to a client to explicitly drop a request
SACK	Sent by the client or the server to ask for missing packets

Table 9.1 – The R2P2 message types

9.2.1 Transport considerations

Figure 9.2 describes a proposed R2P2 header, while Table 9.1 includes the different R2P2 messages. All R2P2 messages are UDP datagrams. R2P2 supports a 16-bit request id whose scope is local to the `(src_ip, src_port)` pair. As such, each client (`(src_ip, src_port)` pair) can have up to 65536 outstanding RPCs, well beyond any practical limitations. The R2P2 header also includes a 16-bit packet id meaning that each R2P2 message can consist of up to 65536 MTU-sized packets. The above two fields can be extended, if necessary, without changing the protocol semantics. Currently R2P2 uses two flags (`F, L`) to denote the first and last packet of a request.

Finally, the R2P2 header contains a `Policy` field, which allows client applications to directly specify certain policies to the router, or any other intermediate middlebox, for this specific RPC. Currently, the only implemented policies are `unrestricted`, which allows the router to direct `REQ0` packet to any worker in the set, and `sticky`, which forces the router to direct the message to the master worker among the set. This mechanism is central to our implementation of a tail-tolerant Redis, based on a master/slave architecture. It is used to direct writes to the master, but balances reads according to the load balancing policy. Additional policies, *e.g.*, session-stickiness, or policies implementing different consistency models, can be implemented in R2P2 middleboxes and will be identified by this header field, thus showcasing the benefits of R2P2's in-network RPC awareness.

Deployment assumptions: We assume that R2P2 is deployed within a datacenter, *i.e.*, the clients, router and servers are connected by a high-bandwidth, low-latency Ethernet fabric. We make no assumptions about the core network that can depend either on ECMP flow hashing or

packet spraying [255, 125, 143]. R2P2 tolerates packet reordering within the same message and reconstructs the message at the end-point. By design, though, there is no ordering guarantee across RPCs, even if they are sent by the same client.

Timer management: Given that the assumed deployment model allows for packet reordering, packet loss detection depends on timers. There is one retransmission timeout `RTO` timer used for multi-packet requests or responses. It is in the order of milliseconds and triggers the transmission of a `SACK` message request for the missing packets. Servers garbage collect RPCs with failed multi-packet requests or multi-packet replies after a few `RTO`s. On the client side there is a timer set by the client application when sending the request. This timer is disarmed when the whole reply is received, and can be as aggressive as the application SLO. Based on this timer applications can implement tail-mitigation techniques [80] or early drop requests based on their importance.

Congestion management: R2P2 focuses on in-network policy enforcement for RPCs; we do not make any explicit contribution in congestion control. Instead, R2P2 can utilize existing solutions for congestion control, including (1) Homa [255], whose message semantics easily map to R2P2's request-response semantics and (2) ECN-based schemes such as DCTCP [7] and DCQCN [383]. Congestion control will be necessary only for multi-packet requests and replies (`REQN` and `REPLY`), and is independent of the interactions described in Fig 9.1. We explored the possibility of an R2P2-specific congestion control scheme that takes advantage of the asymmetry between requests and responses in the context of a master thesis [297]. A system implementation and evaluation of the proposed scheme is left for future work.

Flow Control: R2P2 implements two levels of flow control, one between the client and the middlebox and one between the middlebox and the servers. R2P2 middleboxes can drop individual requests, either randomly or based on certain priority policies, if they become congested, without affecting other requests, thus implementing the first level of flow control. Based on the functionality and the policy, the middlebox is in charge of implementing the second level of flow control to the servers. In the `JBSQ` case, `JBSQ` limits the number of outstanding requests on each server, thus servers can not be overwhelmed.

9.2.2 API

R2P2 exposes a non-POSIX API specifically designed for RPC workloads. Making RPCs first class citizens and exposing the request-response abstraction through the networking stack significantly simplifies writing client-server applications. Application code that traditionally implements the RPC logic on top of a byte stream abstraction is now part of the R2P2 layer of the networking stack.

Table 9.2 summarizes the corresponding application calls and callbacks for the client and

Application Calls	
Type	**Description**
r2p2_poll	Poll for new req/resp
r2p2_send_req	Send a request
r2p2_send_response	Send a response
r2p2_message_done	Deallocate a message

Callbacks	
Type	**Description**
req_recv	Received a new request
req_success	Request was successful
req_timeout	Timer expired
req_error	Error condition

Table 9.2 – The `r2p2-lib` API

server application. The API has an asynchronous design that allows applications to easily send and receive independent RPCs. When calling `r2p2_send_req` the client application sets the timer timeout and callback functions independently for each RPC request. The client and server applications are notified only when the entire response or request messages have arrived through the `req_success` and `req_recv` callbacks, equivalently.

9.3 Implementation

We implement `r2p2-lib` as userspace Linux library on top of either UDP sockets or DPDK [88]. The library links into both client and server application code. It exposes the previously described API and abstracts the differences between the Linux socket and the DPDK-based implementations. The current implementation is non-blocking and `rpc_poll` is typically called in a spin loop. To do so, we depend on `epoll` for Linux, while for DPDK we implemented a thin ARP, IP, and UDP layer on top of DPDK's polling mode driver, and exposed that to `r2p2-lib`. Our C implementation of `r2p2-lib` consists of 1300 SLOC.

R2P2 does not impose any threading model. Given the callback-based design, threads in charge of sending or receiving RPCs operate in a polling loop mode. The library supports symmetric models, where threads are in charge of both network and application processing, by having each thread manage and expose a distinct worker queue through a specific UDP destination port. The DPDK implementation further manages a distinct Tx and Rx queue per thread, and uses Flow Director [112] to steer traffic based on the UDP destination port. In an asymmetric model, a single dispatcher thread links with `r2p2-lib`, and the other worker threads are in charge of application processing only. This model exposes one worker queue via one UDP destination port.

9.4 Related work

RPCs can be transported by different IP-based protocols including HTTP2 [35], QUIC [211], SCTP [324], DCCP [204], or similar research approaches [119, 9, 255, 125, 143] that identify the TCP limitations and optimize for flow-completion time. Libraries such as gRPC [139] and Thrift [333] abstract away the underlying transport stream into request-reply pairs. Approaches such as eRPC [179] aim at end-host system optimizations and are orthogonal to R2P2.

R2P2 adheres and encourages the in-network compute research path by increasing the network visibility to application logic and implementing in-network scheduling. Approaches leveraging in-network compute include caching [171, 224], replicated storage [170], network sequencing [222, 221], DNN training [312, 313], and database acceleration [217].

9.5 Chapter Summary

We revisit the requirements to support µs-scale RPCs across tiers of web-scale applications and propose to solve the problem in the network by making RPCs true first-class citizens of the datacenter. We design, implement and evaluate a proof-of-concept transport protocol developed specifically for µs-scale RPCs that exposes the RPC abstraction to the network and at the endpoints. In the next chapters we we showcase the benefits of the new design by implementing mechanisms for tail-tolerant and fault-tolerant µs-scale RPCs in the network.

10 In-network RPC Load Balancing with JBSQ

10.1 Introduction

In Section 2.5 we described how to distribute request load among different backend-servers. In one approach, clients communicate with the servers through a L4-load balancer. This, though, leads load imbalance and head-of-line blocking, thus bad tail latency. Another connection-oriented approach and similar in terms of tail-latency is to use direct connections and connection pools. In this scheme, clients randomly select a connection from the connection pool. Both of these approaches are equivalent to random load balancing. A theoretically better approach uses a L7 load balancer or reverse proxy [108, 4, 65] to select among replicas on a per request basis, *e.g.,* using a Round-Robin or Join-Shortest-Queue (JSQ) algorithm. Such load balancing policies improve upon random selection, but they do not eliminate head-of-line blocking, while the load balancer can become a scalability bottleneck.

As a first use case for R2P2, we show how to use our network protocol to implement efficient, scalable, tail-tolerant, high-throughput routing of RPCs by introducing *Join-Bounded-Shortest-Queue* (`JBSQ(n)`), a new RPC scheduling policy that splits queues between the router and the servers, allowing only a bounded number of outstanding requests per server, which significantly improves tail-latency. Our design includes an RPC router that can be implemented efficiently either in software or within a programmable switch ASIC such as P4 [39].

In this chapter we describe the following:

- The design of `JBSQ(n)`, a split-queue scheduling policy that utilizes a single in-network queue and bounded server queues and improves tail-latency even for µs-scale tasks.
- The implementation of the R2P2 router on a software middlebox that adds only 5µs to end-to-end unloaded latency and is capable of load balancing incoming RPCs at line rate using only 2 cores.
- The implementation of the R2P2 router within a P4-programmable Tofino dataplane ASIC, which eliminates the I/O bottlenecks of a software middlebox and reduces latency

overhead to 1µs.

Our evaluation, using a range of microbenchmarks, shows that the protocol is suitable for µs-scale RPCs and that its tail latency outperforms both random selection and classic HTTP reverse proxies. The P4-based implementation of R2P2 on a Tofino ASIC adds less than 1µs of latency whereas the software middlebox implementation adds 5µs latency and requires only two CPU cores to route RPCs at 10 Gbps line-rate. R2P2 improves the tail latency of web index searching on a cluster of 16 workers operating at 50% of capacity by 5.7× over NGINX. R2P2 improves the throughput of the Redis key-value store on a 4-node cluster with master/slave replication for a tail-latency service-level objective of 200µs by more than 4.8× vs. vanilla Redis.

10.2 Queueing Analysis

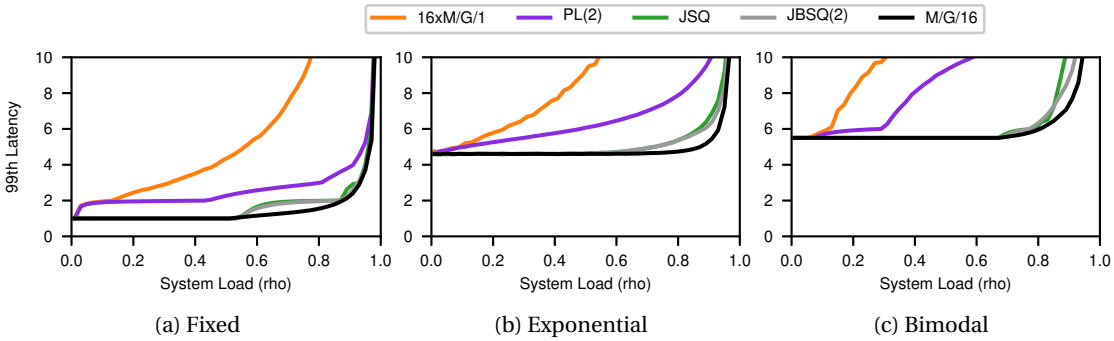
We approach the problem of RPC load balancing from a theoretical point of view by abstracting away system aspects using basic queuing theory. The request-level load balancing policies described in Section 2.5 can be split in two main categories: *push-based* and *pull-based* policies. In push-based policies there is no queue at the load balancer and for every incoming request the load balancer picks a destination and forwards the request. In pull-based policies all incoming requests are queued up in the load balancer before a server asks for a new request. We show the benefits of rich request-level load balancing policies over random-selection among distributed queues (which is equivalent to L4 load balancing) in improving tail-latency through discrete event simulations.

Fortunately, the theoretical answers are clear: single-queue, multi-worker models (*i.e.*, M/G/k according to Kendall's notation) perform better than distributed multi-queue models (*i.e.*, k×M/G/1, with one queue per worker) because they are work-conserving and guarantee that requests are processed in order [214, 366].

Between those two extremes, there are other models that improve upon random selection and are practically implementable through L7 load balancing. Power-of-two [253] (PL(2)), or similar schemes, are still in the realm of randomized load balancing, but perform better than a blind random selection. JSQ performs close to a single queue model for low-variability service times [232].

Figure 10.1 quantifies the tail-latency benefit, at the 99th percentile, for these queuing models observed in a discrete event simulation. We evaluate a configuration with a Poisson arrival process, $k = 16$ workers, and three well-known distributions with the same service time $\bar{S} = 1$. These distributions are: deterministic, exponential and bimodal-1 (where 90% of requests execute in .5 and 10% in 5.5 units) [232].

From the simulation results, we conclude that: (1) there is a dramatic gap in performance between the random, multi-queue model and the single-queue approach, which is optimal



(a) Fixed (b) Exponential (c) Bimodal

Figure 10.1 – Simulation results for the 99th percentile latency across 3 service time distributions with $\overline{S} = 1$

among FCFS queuing systems. (There is no universally optimal scheduling strategy for tail-latency [366].) (2) `PL(2)` improves upon random selection, but these benefits diminish as service time variability increases. `JSQ` performs close to the optimal for low service time variability.

10.3 The Join-Bounded-Shortest-Queue Policy

We define Join-Bounded-Shortest-Queue JBSQ(n) as a policy that can achieve some of the benefits of pull-based policies and reduce head of line blocking without the need for such high coupling. JBSQ splits queues between a centralized component with an unbounded queue and distributed bounded queues of maximum depth n for each worker (including the task currently processed). Once a new request arrives at the load balancer, if there are available slots in some of the server queues, there are less than n outstanding requests, the load balancer picks the queue with the least requests and sends the new request. If all queues are full, the request is queued in a central queue in the load balancer waiting for a queue slot to open. So, JBSQ reduces the chances of a short request being blocked behind a long request since there can only be n-1 delayed requests per queue. Also, JBSQ, unlike the other policies, it can be parametrized according to the application needs, the communication delays in the deployment environment, and the application service time. The single-queue model is equivalent to `JBSQ(1)` whereas `JSQ` is approximately equivalent to `JBSQ(`∞`)`.

In Figure 10.1 we also plot `JBSQ(2)`. We observe that JBSQ(2), although it deviates from the single queue model, outperforms JSQ under high load as the service time variability increases. As we increase n the `JBSQ(n)` curve deviates more from single queue.

These results are purely theoretical and in particular assume perfect global knowledge by the scheduler or load balancer. This global view would be the result of communication between the workers and the load balancer in a real deployment. Any practical system must consider I/O bottlenecks and additional scheduling delays because of this communication. In this chapter, we make the claim that JBSQ(n) can be implemented in a practical system and can

deliver maximal throughput with small values of n even for µs-scale tasks, thus minimizing tail latency and head-of-line blocking.

10.4 JBSQ router design considerations

R2P2 exposes the request-response abstraction to the network as a first-class citizen. It is expected that a software or hardware middlebox will manipulate client requests to implement a certain policy, *e.g.*, scheduling, load balancing, admission control, or even application logic, *e.g.*, routing requests to the right server in a distributed hash table. In this section, we discuss the design choices regarding an R2P2 request router implementing the JBSQ scheduling policy. Similar ideas can be applied to other middleboxes with alternative functionality.

The choice of JBSQ: As seen in Figure 10.1 JSQ and JBSQ perform closer to the optimal single queue model. JBSQ though offers several practical benefits over JSQ. It implements router-servers flow control and can be implemented within a Tofino ASIC. JSQ requires finding the minimum among a number of values, which is hard to implement in a hardware dataplane. Also, JBSQ achieves better latency under high load and service time dispersion. That is because JSQ uses the queue size as a proxy for queuing time, which can be misleading in the presence of service-time dispersion.

R2P2-FEEDBACK messages: To implement the JBSQ(n) policy we leverage the R2P2-FEEDBACK messages provided by the R2P2 specification. These messages, sent by the servers back to the router after completing the service of a request, specify: (i) The maximum number of outstanding RPCs the server is willing to serve (the "n" in JBSQ(n)). By sending the current "n" in every R2P2-FEEDBACK message, servers can dynamically change the number of outstanding requests based on the application SLOs. (ii) The number of requests this server has served including the last request. The router uses this information to track the current number of outstanding requests in the server's bounded queue. This field makes the message itself idempotent and the protocol robust to R2P2-FEEDBACK drops.

We note that this approach puts each server in charge of controlling its own lifecycle by sending unsolicited R2P2-FEEDBACK messages, *e.g.*, to join a load balancing group, leave it, adjust its bounded queue size based on its idle time, or to periodically signal its idleness.

Direct client request - direct server return: R2P2 implements direct server return (DSR) [267, 147] since the replies do not go through the router. This is a widely-used technique in L4 load balancers with static policies [267]. R2P2 uses DSR while implementing request-level load balancing. In addition, R2P2 implements direct client request, where the router handles only the first packet of a multi-packet request, while the rest is streamed directly to the corresponding server, thus avoiding router IO bottlenecks.

Deployment: A software R2P2 router is deployed as a middlebox and traffic is directed to its IP address. The hardware R2P2 router is also deployed as an IP-addressed middlebox. The same hardware can also be a Top-of-Rack switch serving traffic to servers within the rack, following a "rack-scale" deployment pattern. In such a pattern, the router has full visibility on the RPC traffic to the rack and all packets go through the ToR switch. This could enable simplifications to the packet exchange, *e.g.*, using `R2P2-FEEDBACK` messages only for changing the depth of the bounded queues; the ToR can estimate their current size by tracking the traffic.

Router high availability: The router itself is nearly stateless and a highly-available implementation of the router is relatively trivial. Upon a router failure, only soft state regarding the routing policy is lost, including the current size of the per-worker bounded queue and the queue of pending RPCs. Clients simply failover to the backup router using a virtual IP address and reissue RPCs upon timeout, using the exact same mechanism used to handle a `REQ0` packet loss. Servers reconstruct the relationship with the router with their `R2P2-FEEDBACK` message to the new router.

Server membership: Servers behind the R2P2 router can fail and new servers can join the load balancing group. `R2P2-FEEDBACK` messages implicitly confirm to the router that a server is alive. In case of a failure, the lack of `R2P2-FEEDBACK` messages will prevent the router from sending requests to the failed server, and the bounded nature of `JBSQ(n)` limits the number of affected RPCs. Similarly, newly-added servers can send `R2P2-FEEDBACK` messages to the router informing about their availability to serve requests.

The choice of JBSQ(n): The choice of `n` in `JBSQ` is crucial. A small `n` will behave closer to a single-queue model, but will restrict throughput. The rationale behind the choice of `n` is similar to the Bandwidth Delay Product. On each queue there should be enough outstanding requests so that the server does not stay idle during the server-router communication. For example, for a communication delay of around 15 µs and a fixed service time of 10 µs, `n=3` is enough to achieve full throughput. Shorter service times will require higher `n` values. High service time dispersion and batching on the server will also require higher values than what predicted by the heuristic. Servers can even dynamically adjust the value of `n` based on their processing rate and minimal idle time between requests.

10.5 Implementation

We provide two implementations for the `JBSQ` policy on the R2P2 router: one based on software R2P2 of DPDK (Subsection 10.5.1) and one using the $P4_{14}$ programming language [283] to run within a Barefoot Tofino ASIC [28] (Subsection 10.5.2). We also implement other simpler RPC-level load balancing policies that we use as a baseline in our evaluation.

10.5.1 Router - software implementation

We implemented a `Random`, a `Round-Robin`, a `JSQ` and a `JBSQ(n)` policy on the software router. The main implementation requirements for the router are (1) it should add the minimum possible latency overhead, and (2) it should be able to process short `REQ0` and `R2P2-FEEDBACK` messages at line rate. While the router processes only those two types of packets, the order in which it processes them matters. Specifically for `JBSQ`, the ideal design separates `REQ0` from `R2P2-FEEDBACK` messages into two distinct ingress queues and processes `R2P2-FEEDBACK`s with higher priority to ensure that the server state information is up-to-date and minimize queuing delays.

Our DPDK implementation uses two different UDP ports, one for each message type, using Flow Director for queue separation. Given the strict priority of control messages and the focus on scalability, we chose a multi-threaded router implementation with split roles for `REQ0` threads and `R2P2-FEEDBACK` threads, with each thread having its own Rx and Tx queues.

`JBSQ(n)` requires a counter per worker queue that counts the outstanding requests. To minimize cache-coherency traffic, the router maintains two single-writer arrays, one updated on every `REQ0` and the other on every `R2P2-FEEDBACK`, with one entry per worker.

The implementation of the `R2P2-FEEDBACK` thread is computationally very cheap and embarrassingly scalable. Processing `REQ0` messages requires further optimizations to reduce cache-coherency traffic, *e.g.*, maintain the list of known idle workers, cache the current queue sizes, *etc.* Our implementation relies on adaptive bounded batching [34] to amortize the cost of PCIe I/O operations, as well as that of the cache-coherency traffic (the counters are read once per iteration). We limit the batch size to 64 packets.

Finally, we implement a tweak to the `JBSQ(n)` policy with $n \geq 2$: when no idle workers are present, up to 32 packets are held back for a bounded amount of time on the optimistic view that an `R2P2-ACK` message may announce the next idle worker. This optimization helps absorb instantaneous congestion and approximate the single-queue semantics in medium load situations.

10.5.2 P4/Tofino implementation

We built a proof-of-concept P4 implementations of R2P2 router for Tofino [28] using $P4_{14}$ [283]. Similar to the software implementation, the switch only processes `REQ0` and `R2P2-FEEDBACK` messages and leverages P4 registers to keep soft state. P4 registers are locations in the ASIC SRAM, which can be read and updated from both the control and dataplane.

We focus our description on the implementation of `JBSQ(n)` for the Tofino dataplane, as the others are trivial in comparison. It consists of 480 lines of P4 source, including header descriptions. Unlike the software implementation that can easily buffer the outstanding `REQ0` messages if there is no available server queue, high-performance pipelined architectures,

such as Tofino, do not allow buffering in the dataplane. Thus, our P4 logic executes as part of the ingress pipeline of the switch and relies heavily on the ability to recirculate packets through the dataplane via a virtual port. The implementation leverages an additional header that is added to the packet to carry metadata through the various recirculation rounds and is removed before forwarding the packet to the target server.

The logic for `REQ0` tries to find a server with $\leq i$ outstanding packets in round i. There is one register instance corresponding to each server, holding the number of outstanding requests. If a suitable server is found, the register value is increased by one, the packet destination is changed to the address of the equivalent server, and the packet is directed to the egress port. We start with $i = 0$ and we increase till $i = n$ from `JBSQ(n)`. When `i` reaches `n` and there is still no available server, we keep recirculating the packet without increasing `i` further. As an optimization to reduce the number of recirculations, the dataplane keeps the `i` for the last forwarded request and starts from that.

To overcome the Tofino limitation of only being able to compare a limited number of registers in one pass, we also leverage recirculation to inspect the outstanding requests of each bounded queue in each round. Register instances that correspond to different queues are organized in groups that can be checked in one pass. If no available queue is found in the first group, the packet is recirculated (without increasing `i`) and the second group of queues is checked, *etc.* When a `REQ0` arrives, it is initially assigned to a group in a round-robin fashion to further reduce the amount of recirculations.

The logic for `R2P2-FEEDBACK` decrements the outstanding count for the specific server based on the packet source and consumes the packet without forwarding it.

The use of recirculation has two side-effects: (1) the order of RPCs cannot be guaranteed as one packet may be recirculated while another one is not; (2) the atomicity of the full set of comparisons is not guaranteed as `R2P2-FEEDACK` packet may be processed while an `REQ0` packet is being recirculated. Non-optimal decisions may occur as the result of this race condition.

10.6 Evaluation

To evaluate the performance and the efficacy of the R2P2 protocol, the two implementations of the router, as well as the trade-offs in using `JBSQ(n)` over other routing policies, we run a series of synthetic microbenchmarks and two real applications in a distributed setup with multiple servers. The microbenchmarks depend on an RPC service with configurable service time and response size. All our experiments are open-loop [315] and clients generate requests with a Poisson inter-arrival time. We use two baselines and compare them against different configurations for R2P2 with and without the router: (1) vanilla NGINX [269] serving as reverse proxy for HTTP requests; and (2) ZygOS [299], a state-of-the-art work-conserving multicore scheduler. As a load generator we use an early version of Lancet [201].

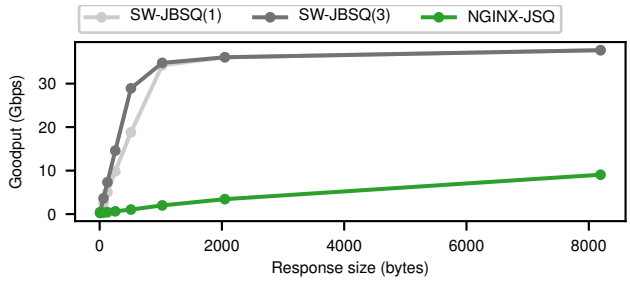


Figure 10.2 – Achieved Goodput as a function of the response size for the JBSQ policy on the software router managing 4 servers connected with 10GbE NICs compared to NGINX configured as HTTP reverse proxy loadbalancing the same 4 servers using a JSQ policy.

Our experimental setup consists of cluster of 17 machines connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The machines are a mix of Xeon E5-2637 @ 3.5 GHz with 8 cores (16 hyperthreads), and Xeon E5-2650 @ 2.6 GHz with 16 cores (32 hyperthreads). All machines are configured with Intel x520 10GbE NICs (82599EB chipset). To reduce latency and jitter, we configured the machine that measures latency to direct all UDP packets to the same NIC queue via Flow Director. The Barefoot Tofino ASIC runs within a Edgecore Wedge100BF-32X. The Edgecore is directly connected to the Quanta switch via a 40Gbps link and therefore operates as a 1-port router.

10.6.1 Router characterization

We use the synthetic RPC service to evaluate the latency overhead of the router, the maximal throughput and the optimal request load balancing policy. We configure a setup of 4 servers with 16 threads (64 independent workers), running the synthetic RPC service over DPDK.

Throughput: We first evaluate the sustainable throughput of the software router. We run a synthetic RPC service with 8-byte requests and we configure the size of the response.

Figure 10.2 shows the achieved goodput as a function of the response size, and compares a configuration with R2P2 messages handled by a JBSQ load balancing policy, with a NGINX configured as reverse proxy for HTTP messages. For small response sizes, the router is bottlenecked by the router's NIC's packets per second (PPS), or the number of outstanding requests in each queue, n in $JBSQ(n)$. JBSQ(3) was enough to achieve maximum throughput. As the response size increases though, the application goodput converges to $4 \times$ 10GbE, the NIC bottleneck of the 4 servers with payloads as small as 2048. Obviously, this is made possible by the protocol itself, which bypasses the router for all REPLY messages. Note that because R2P2 leverages both Direct Server Return and Direct Client Request, even in cases of large requests the router would not be the bottleneck, unlike traditional L4 DSR-enabled load balancing. In contrast, the NGINX I/O bottleneck limits goodput to the load balancer's 10Gbps NIC.

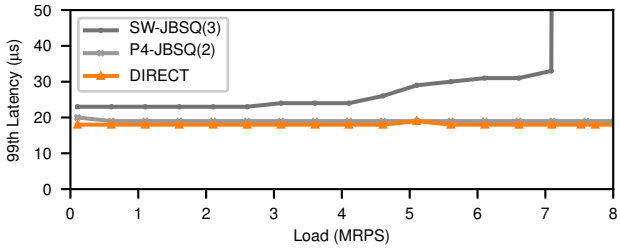


Figure 10.3 – Tail-latency as a function of throughput for a zero service time synthetic RPC service for the software (`SW-JBSQ`) and the Tofino (`P4-JBSQ`) implementation of JBSQ compared to `DIRECT`. In `DIRECT` clients bypass the router and talk directly to servers by making a random server choice.

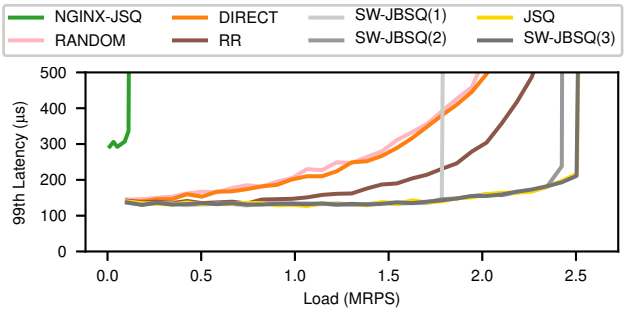


Figure 10.4 – Evaluation of different load balancing policies for an exponential service time workload with $\bar{S} = 25\mu s$.

Latency overheads and saturation: Figure 10.3 uses a zero-cost ("echo") RPC service with 8-byte requests and responses, to measure the 99th percentile tail latency as a function of the load for the software middlebox and the Tofino router with the `JBSQ` policy. As a baseline, we use a `DIRECT` configuration where clients bypass the router and send requests directly to the servers after a random choice. The figure shows that the latency added by the router is 5µs for the software middlebox and 1µs for the Tofino solution. The software latency is consistent with the characteristics of one-way forwarding performance of the Intel x520 chipset using DPDK. The hardware latency is consistent with the behavior of an ASIC solution that processes and rewrites packet headers in the dataplane. Figure 10.3 also shows the point of saturation, which corresponds to 7 MRPS for the software middlebox. Given that for every request forwarded the router receives one `R2P2-FEEDBACK` message, the router handles more than 14M PPS, which is the hardware limit. We were unable to characterize the maximal per-port capability of the Tofino ASIC running the R2P2 logic beyond >8 MRPPS with tiny requests and replies, simply for lack of available client machines. We also observe that the hardware implementation, as expected, requires a smaller `n` for `JBSQ(n)`. In the figure we show the smallest value of `n` that achieved maximum throughput.

Comparison of scheduling policies: Figure 10.4 uses a synthetic $\bar{S} = 25\mu s$ workload to evaluate the different request load balancing policies, implemented on the software router. We

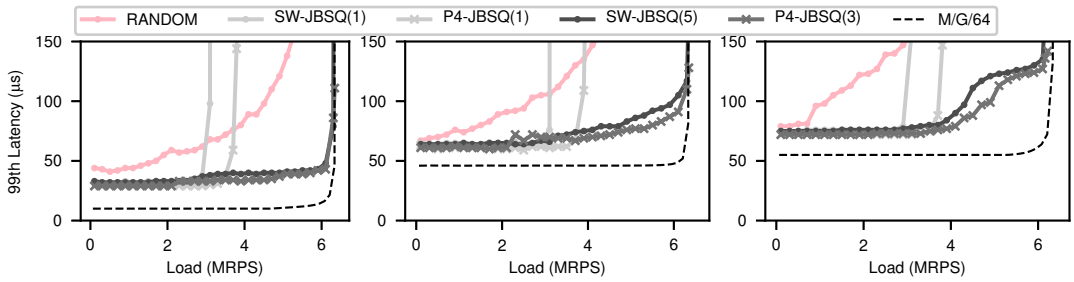
evaluate the following policies: DIRECT, where clients bypass the router by making a random server selection, RANDOM where clients talk to the router and the router makes a random selection among the servers, RR where the router selects a target server in a round-robin manner, SW-JBSQ(n) which is the software implementation for the bounded shortest queue with n outstanding requests, and JSQ which is the R2P2 router's implementation of the join-shortest-queue policy. We also compare R2P2 with using NGINX as an HTTP reverse proxy implementing a JSQ policy, which is a vanilla, widely-used deployment for request-level load balancing.

We make the following observations: (i) NGINX overheads prevent throughput scalability; (ii) DIRECT and RAND configurations perform similarly for R2P2, which is the result of a random choice (in the client or the router equivalently); (iii) RR performs better than random choice, but worse than JBSQ, given the service time dispersion; (iv) JBSQ($n \geq 3$) achieves maximum throughput. Given that the communication time between the server and the router is ~ 15µs and the exponential service time dispersion, this is on par with our analysis in § 10.4. (v) JSQ performs similarly to JBSQ(3) for this service time.

10.6.2 Synthetic Time Microbenchmarks

Figure 10.5 evaluates JBSQ(n) performance with an aggressive $\bar{S} = 10$µs mean service time and three different service time distributions: Fixed, Exponential and Bimodal where 10% of the request are 10x slower than the rest [232]. We present results for both the software and Tofino implementation, for JBSQ(1) and the optimal n choice for each configuration. Requests and the responses are 8 bytes. We observe:

- For all experiments, all JBSQ(n) variants approximate the optimal single-queue approach ($M/G/64$) until the saturation point for JBSQ(1).
- Beyond the saturation point of JBSQ(1), an increase in the tail latency as the system configuration trades off higher throughput (*i.e.,* JBSQ($n > 1$)) against the use of a theoretically-optimal approach.
- A comparison between the software and hardware implementation shows that more outstanding requests are required for the software implementation; this is because the communication latency between the server and the hardware router is ~5µs faster.
- JBSQ achieves the optimal performance, as predicted by the M/G/64 model, both for the software and the hardware implementation within the 150µs SLO.
- Reducing n can have a considerable impact on tail-latency especially in cases with high service time dispersion, as it can be seen in Figure 10.5c (SW-JBSQ(5) vs. P4-JBSQ(3))



(a) Fixed (64 workers) (b) Exponential (64 workers) (c) Bimodal (64 workers)

Figure 10.5 – Synthetic Service Time Microbenchmarks. Service time $\bar{S} = 10\mu s$.

10.6.3 Multi-packet Requests Microbenchmark

R2P2 implements the following logic in splitting requests to packets. If the request fits in a single packet, the whole request payload is transferred with `REQ0`. In the case of a multi-packet request, `REQ0` is a 64-byte packet, carrying only the first part of the request and the rest of the payload is transferred with the `REQN` packets directly to the server. This way the router does not become a throughput bottleneck in the case of large requests, while the extra round-trip is avoided in the case of small requests.

To evaluate the extra round-trip that R2P2 introduces in the case of multi-packet requests with the distinction between `REQ0` and `REQN`, we ran a synthetic microbenchmark with larger requests. Based on the above logic, a 1464-byte request is the biggest request that fits in a single packet given the size of protocol headers. Equivalently, a 1465-byte request is the smallest request that requires 2 packets, and consequently an extra round-trip. We run the synthetic service time RPC server with the bimodal service time distribution of $\bar{S} = 10$ and the 2 different request sizes. We compare the `DIRECT` deployment with one using the router with the `JBSQ` policy.

Figure 10.6 summarizes the result of the experiment. We observe that there is a fixed gap of around 15µs between `DIRECT-1464` and `DIRECT-1465` curves that corresponds to the extra round-trip between the client and the server. We, also, run the multi-packet request scenario while using the P4 router with the `JBSQ` policy. We show that despite the extra round-trip, the intermediate hop, and the increased number of packets to process, the 99th percentile latency is close to the single-packet scenario in the `DIRECT` case, which justifies our design decision to pay an extra round-trip to achieve better scheduling.

10.6.4 Using R2P2 for server work conservation

We now demonstrate how the use of network-based load balancing, *e.g.*, using R2P2, can increase the efficiency of a *single* server scheduling tasks. For this, we compare R2P2 with `JBSQ` with the performance of ZygOS [299], a state-of-the-art system optimized for µs-scale,

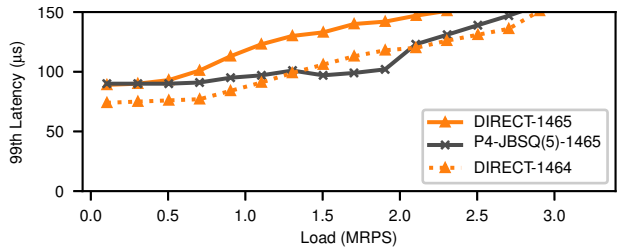


Figure 10.6 – Bimodal service time with $\bar{S}$ = 10µs and 64 workers with single and multi-packet requests. `DIRECT-1464` corresponds to an 1-packet request, while `DIRECT-1465` and `P4-JBSQ(5)` correspond to 2-packet requests.

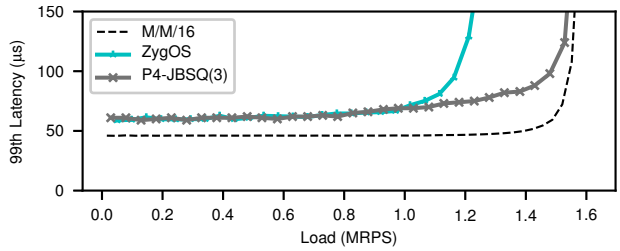


Figure 10.7 – Comparison of R2P2 with the ZygOS [299] work-conserving scheduler: Exponential workload with $\bar{S}$ = 10µs.

multicore computing that includes a work-conserving scheduler within a specialized operating system. ZygOS relies on work-stealing across idle cores and makes heavy use of inter-processor interrupts. Both ZygOS and `JBSQ(n)` offer a work-conserving solution to dispatch requests across the multiple cores of a server: ZygOS does it within the server in a protocol-agnostic manner, whereas R2P2 implements the policy in the network.

Figure 10.7 compares ZygOS with the Tofino implementation of `JBSQ(3)` for the 10µs exponentially-distributed service time workload using a single Xeon server. As in the previous configurations, for the R2P2 implementation each of the 16 Xeon cores, is exposed as a worker with a distinct queue to the router, listening to a different UDP port. In this experiment, the theoretical lower bound is therefore determined by $M/M/16$. We observe that `JBSQ(3)` exceeds the throughput performance of ZygOS, with no visible impact on tail latency despite the additional hop and that `JBSQ(3)` is sufficient to achieves the maximum throughput. For a service-level objective set at 150µs, R2P2 with `JBSQ(3)` outperforms ZygOS by 1.26×. The explanation is that the R2P2 server operates on a set of cores in parallel without synchronization or cache misses, whereas ZygOS has higher overheads due to protocol processing, boundary crossings, task stealing, and inter-processor interrupts.

10.6.5 Lucene++

Web search is a replicated, read-only workload with variability in the service time coming from the different query types, thus it is an ideal use-case for `R2P2-JBSQ`. For our experiments we

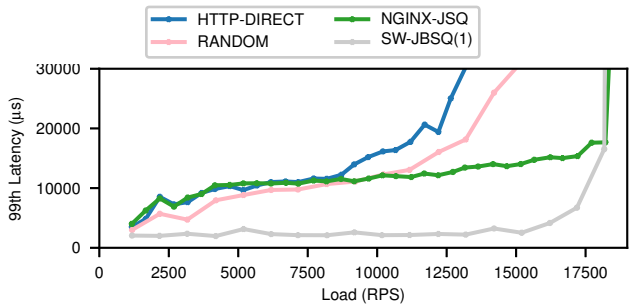
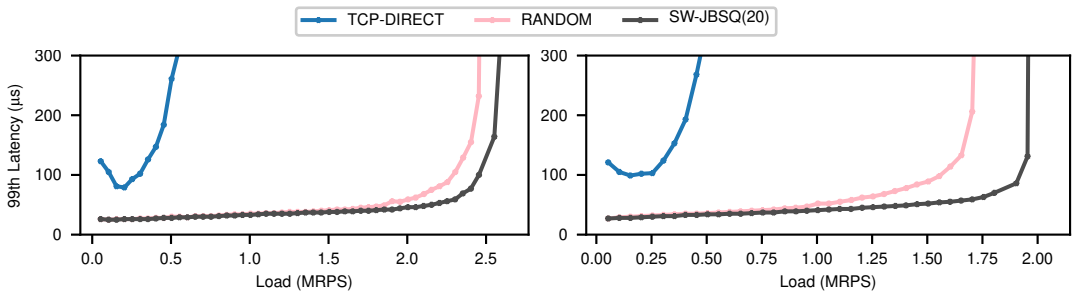


Figure 10.8 – Lucene++ running on 16 16-threaded workers



(a) Standard USR workload [14] (0.2% writes)

(b) Modified USR with 2% writes

Figure 10.9 – 99th percentile latency vs. throughput for Redis in a 4-node master/slave configuration.

used Lucene++ [233], which is a search library ported to serve queries via either HTTP or R2P2. A single I/O thread dispatches one request at a time to 16 Lucene++ worker threads, each of them searching part of the dataset. The experimental setup relies on 16 disjoint indices created from the English Wikipedia page articles dump [367], yielding an aggregated index size of 3.5MB. All indices are loaded in memory at the beginning of the execution to avoid disk accesses. The experimental workload is a subset of the Lucene nightly regression query list, with 10K queries that comprise of simple term, Boolean combinations of terms, proximity, and wildcard queries [234]. The median query service time is 750µs, with short requests taking less than 450µs and long ones over 10ms.

Figure 10.8 summarizes the experiment results for running Lucene++ on a 16-server cluster, each using 16 threads. The `NGINX-JSQ` and `HTTP-DIRECT` experiments rely on 1568 persistent TCP client connections. First, we observe that `HTTP-DIRECT` over TCP and `RANDOM` over R2P2 which are multi-queue models, have higher tail-latency. Then, we see that `NGINX-JSQ` and `SW-JBSQ(1)` on R2P2 deliver the same throughput; system and network protocol overheads are irrelevant for such coarse-grain workload. Also, $n = 1$ is enough to get maximum throughput, given the longer average service time. `SW-JBSQ(1)` delivers that throughput via the optimal single-queue implementation, with a significant impact on tail latency. As a result, R2P2 lowers the 99th percentile latency by 5.7× at 50% system load over `nginx`.

10.6.6 Redis

Redis [304] supports a master/slave replication scheme with read-only slaves. We ported Redis on R2P2 and ran it on DPDK for the Facebook USR workload [14]. We used the `sticky` R2P2 policy (see Section 9.2) to direct writes to the master node and we load balance reads across the master and slave nodes, based on the `RANDOM` and the `JBSQ` policy. Redis has sub-µs service times. Thus, to achieve maximum throughput we had to increase the number of tokens to 20 per worker (`SW-JBSQ(20)`), for the software router. For the vanilla Redis over TCP clients randomly select one of the servers for read requests, while they only send write requests to the master.

Figure 10.9a shows that R2P2, for an SLO of 200µs at the 99th percentile, achieves 5.30× better throughput for the USR workload over vanilla Redis over TCP (`TCP-DIRECT`) because of reduced protocol and system overheads, while `SW-JBSQ(20)` achieves slightly better throughput than `RANDOM` for the same SLO. Figure 10.9b increases the write percentage of the workload from 0.2% to 2%, which increases service time variability: R2P2 `RANDOM` has 4.09× better throughput than `TCP-DIRECT`. `SW-JBSQ(20)` further improves throughput by 18%, for a total speedup of 4.8×, as a result of better load balancing decisions.

10.7 Related work

Load dispatching, direct or through load balancers, typically *pushes* requests to workers, requiring tail-mitigation techniques [80, 145]. Load balancers proxy RPC protocols such as HTTP in software [269, 96, 275] or in hardware [4, 108, 65, 251]. In Join-Idle-Queue [232], workers *pull* requests whenever they are idle. R2P2 exposes the RPC abstraction to the network to achieve better RPC scheduling, and to the application to hide the complexity of the underlying transport. R2P2 additionally supports `JBSQ(n)`, which exposes the tradeoff between maximal throughput and minimal tail latency explicitly.

Task scheduling in distributed big data systems is largely aimed at taming tail-latency and sometimes depends on split-queue designs [374, 282, 185, 303, 82, 83, 296], typically operating with millisecond-scale or larger tasks. R2P2 provides the foundation for scheduling of µs-scale tasks.

Multi-core servers are themselves distributed systems with scheduling and load balancing requirements. This is done by distributing flows using NIC mechanisms [308] in combination with operating systems [289, 100] or dataplane [34, 290, 166] support. Zygos [299] and Shinjuku [178] are an intra-server, work-conserving schedulers for short tasks that rely on task stealing and inter-processor interrupts. R2P2 eliminates the need for complex task stealing strategies by centralizing the logic in the router.

Recent work has focused on key-value stores [281, 250, 304, 229]. MICA provide concurrent-read/exclusive-access (CREW) within a server [229] by offloading the routing decisions to

the client, while hardware and software middleboxes [171, 224, 271] or SDN switches [62, 41] enhance the performance and functionality of key-value stores in-network. RackOut extended the notion of CREW to rack-scale systems [272]. R2P2 supports general-purpose RPCs not limited to key-value stores, together with a mechanisms for steering policies which can be used to implement CREW both within a single server and across the datacenter.

10.8 Chapter Summary

We showcase the benefits of the R2P2 protocol by implementing efficient, tail-tolerant µs-scale RPC load-balancing based on a software router or a programmable P4 ASIC. Our in-network RPC lod balancing approach outperforms standard load balancing proxies by an order of magnitude in throughput and latency, achieves close to the theoretical optimal behavior for 10µs tasks, reduces the tail latency of websearch by 5.7× at 50% load, and increases the scalability of Redis in a master-slave configuration by more than 4.8×.

11 In-network SLO-Aware RPC Flow Control

11.1 Introduction

Datacenter applications, such as web-search, e-commerce, social-networking, *etc.* need to operate under strict Service-Level-Objectives (SLO) for their tail-latency. Complying or violating those SLOs instantly reflects on user satisfaction and engagement. Thus, there is a tremendous on-going effort both in academia and industry in building low-latency systems for microsecond-scale computing [31].

Single-digit round-trip times inside a datacenter are considered commonplace with recent advancements in hardware and software. Dataplane operating systems, such as IX [34], and Arrakis [290] and kernel-bypassing techniques have reduced the system overheads present in commodity operating systems. µs-scale schedulers such as ZygOS [299], and Shinjuku [178] managed to reduce tail-latency even further compared to dataplanes, with smarter scheduling. Systems such as Shenango [280] brought efficiency while maintaining the performance benefits. On the networking side, datacenter-specific congestion control algorithms, such as Homa [255], deal with tail behaviours coming from in-network delays by almost eliminating in-network queueing. However, all of the previous efforts approach tail-tolerance as a metric, and report the achieved throughput at a specific latency SLO in steady state when the offered load is below the system capacity, ignoring the system behaviour under more realistic unpredictable conditions with load bursts.

A tail-tolerant system, on the other hand, is designed to operate according to a specific latency SLO and minimizes the SLO violations under any conditions, trading throughput for predictable behaviour. In this paper we advocate that tail-tolerance should be approached as a system design principle instead of a best-effort metric.

We draw inspiration from research on fault-tolerant systems. The latency of a complex fan-out request is defined by the slowest sub-request [80]. Similarly the mean time to failure of a complex system was defined by the mean time to failure of its most failure-prone sub-component before the advancement in fault-tolerance research. However, reasoning about

fault-tolerance is a ubiquitous part of the system design process today. System designers, leveraging the appropriate form of redundancy, provision for their systems according to the expected risk of failure and can provide explicit availability guarantees. For example, a primary backup [46] system can tolerate f failures out of $f+1$ replicas, state machine replication [207, 276] f out of $2f+1$, while and tandem processes [137] one out of two, *etc.* There is no similar systematic approach to reason about tail behaviours, though.

As a first step towards systematically tail-tolerant systems, we design SVEN (**S**LO **V**iolations **E**limi**N**ator), a system to control the tail behaviour of latency critical services. SVEN operates at the remote procedure call (RPC) level and is part of the RPC transport or library, thus remaining application agnostic. It is orthogonal to any existing scheduling or flow control mechanisms, and can be used at any level inside a fan-out/fan-in application, namely at the aggregator nodes or the leaves. SVEN performs dynamic RPC admission control based on the currently observed latency distribution aiming to keep the incoming load below the threshold that violates the target latency SLO.

We implemented an initial SVEN proof of concept on top of R2P2 [202] that runs on a Barefoot Tofino ASIC [28]. Our evaluation shows that SVEN can identify the load level that violates the latency SLO across a variety of service time distributions without any application-specific configuration, and maintain throughput below that level, even in cases where the in-coming load was above the system capacity. SVEN splits its functionality between the programmable switch's control and dataplane and does not consume any server resources.

11.2 Motivation

Certain datacenter applications can trade-off harvest for yield [120], according to the specific latency SLOs they need to provide. Namely, latency-critical applications might be willing to ignore some of the slow sub-RPCs, and thus reduce the harvest and the quality of their response, in order to reply sooner back to the client. Consider for example web-search: a fast result is more useful than a complete result.

Mechanisms that trade-off harvest for yield can be implemented either proactively or reactively. Previously proposed systems, such as Zeta [151], terminate requests that exceed their given service time, and return partial answers to the client. Such reactive designs are steps towards tail-tolerance, but suffer from two main pitfalls. First, they are application specific — cancelling a request and removing transient state requires application knowledge. Second, they waste resources serving requests that are not used in the final answer instead of serving new requests. Instead, a proactive design, that avoids performing sub-RPCs rather than cancelling them, would reduce the need for application awareness and use resources more efficiently.

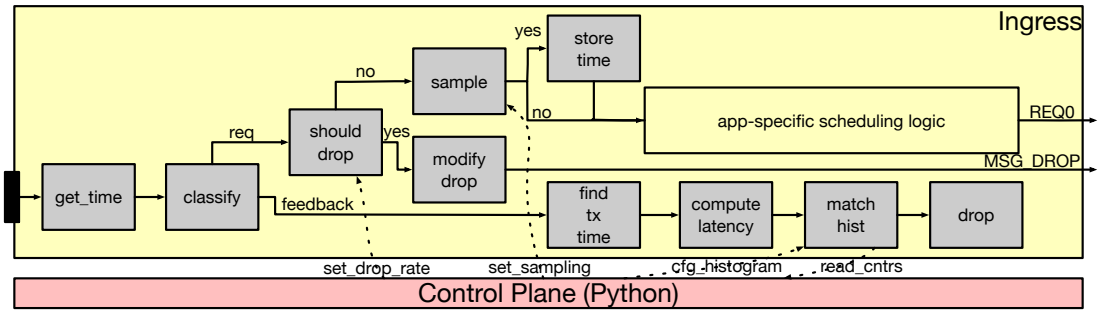


Figure 11.1 – SVEN's dataplane sample implementation running in Tofino's ingress pipeline.

11.3 Design

SVEN is a system designed to reduce the latency SLO violations for RPC services that run inside the datacenter by proactively controlling the incoming load based on an estimation of the current end-to-end latency distribution of the RPC service. Reducing the offered load will reduce queueing and as a result the end-to-end request latency.

We set the following requirements while designing SVEN: i) our solution should be application-agnostic and it should work for different service time distributions without need for re-configuration; ii) it should be independent of and complementary to any scheduling mechanism or policy both at the leaf or the aggregator nodes in a fan-out/fan-in application; iii) it should be implementable on programmable network devices, *e.g.*, P4 switches, to avoid adding any CPU overhead to the RPC servers or latency to the end-to-end result.

We design SVEN as an R2P2 middlebox (Figure 9.1) that can do SLO-aware RPC admission control for several servers. R2P2 [202] revealed the benefits of implementing an in-network RPC scheduling policy that come from the global view of the infrastructure. We place SVEN before the scheduling logic on the same middlebox so that the scheduler deals only with the requests to be executed and not with the ones that got rejected due to admission control. We split SVEN's functionality between a control and a dataplane.

11.3.1 SVEN Dataplane

SVEN's dataplane deals with the `REQ0` packets from the clients and the `FEEDBACK` messages from the servers. It is in charge of dropping incoming requests based on a drop rate defined by the control plane, and it also needs to measure RPC latencies so that the control plane can estimate the current RPC latency distribution.

The dataplane probabilistically decides whether to forward `REQ0` to one of the servers, or drop it, based on the drop rate set by the control plane. In case of a drop, the dataplane sends a `DROP_MSG` back to the client to avoid request timeouts. This way the client early on knows that the request will not be executed and it should not wait for it.

Incoming requests can be dropped uniformly or there can be different drop rates for different request types. If there are different request types, the control plane can set different drop rates per type. R2P2's header defines a message type field, as well as a policy field, that can be used to encode different request types in an application-agnostic manner, so that the control plane can apply the equivalent drop rate. Based on this mechanism each application can set different priorities for its requests corresponding to different drop rates. Also, this way an application can define requests that should not be dropped, *e.g.,* non read-only requests, acknowledging the risk of violating the latency SLO, though.

For the latency estimation, SVEN's dataplane takes timestamps when receiving `REQ0`s from clients, and `FEEDBACK`s from servers that signal a request completion. The difference between the `FEEDBACK` and the `REQ0` timestamps for the same request can be used as a proxy for the latency perceived by the RPC client. SVEN keeps the up-to-date latency distribution in the programmable switch dataplane through a histogram implemented as counters to a match-action table whose cells correspond to the latency histogram buckets. The dataplane can maintain different latency histograms for different request types. In the simplest case, SVEN needs one table, assuming one request type, and two buckets, for requests under and above the SLO.

There are different ways to match the `REQ0` and `FEEDBACK` timestamps to get the latency of a specific request. One approach is to add the `REQ0` timestamp in the `REQ0` packet itself and require the R2P2 stack to echo the timestamp back with the `FEEDBACK` message, similarly to TCP's timestamp option used for RTT estimation. The alternative is to keep the `REQ0` timestamp in the dataplane indexed with R2P2's 3-way tuple of `(src_ip, src_port, req_id)` and include the same tuple in the FEEDBACK message to do the matching. In our implementation we used the latter approach, because it required fewer changes to the R2P2 code base.

The current design assumes that servers send a `FEEDBACK` message for each request. However, this number can be reduced if it affects the server scalability. The middlebox may sample the timestamped RPCs using a specific request type, *e.g.,* timestamped request, so that the server only sends back a `FEEDBACK` message for certain requests.

Figure 11.1 describes the dataplane processing processing pipeline and its interactions with the control plane. Once a packet arrives, the dataplane first take a timestamp to avoid measuring any time spent in the middlebox. Then, there are different paths based on the packet type (`REQ0` or `FEEDBACK`). In the case of `REQ0`, SVEN is completely independent and runs before the application specific scheduling logic, *e.g.,* `JBSQ` or random server selection.

11.3.2 Control plane

SVEN's control plane runs a control loop that computes the RPC drop rate to be applied based on the current latency distribution. The only configuration parameter for this control loop

is the target latency SLO at a specific latency percentile, *e.g.*, 300µs at the 99-th percentile. The input to the control loop is the current estimation for the latency at the current latency percentile, coming from the dataplane. The output of the control loop is the drop-rate that the middlebox needs to apply to the incoming requests in order to reduce latency violations.

There are different ways to design this control plane, such as simple heuristics, control theory, or even online learning. The design and implementation of this control plane is orthogonal to the design of SVEN. In our proof-of-concept implementation we used a simple additive-increase-additive-decrease control over an exponentially weighted average estimation of the target-percentile latency. Listing 11.1 describes the control loop logic and the dataplane-control plane interaction. We acknowledge that this is not an optimal control plane design and it makes many sweeping simplifications, such as ignoring hysterisis, but its sole purpose is to showcase the usability of our design.

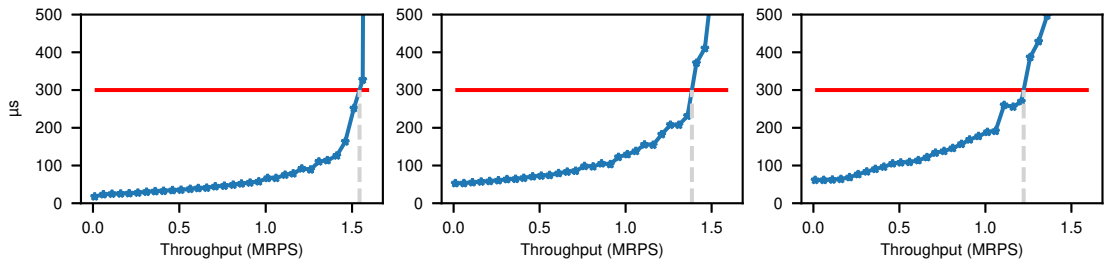
```
STEP = 0.01
SLEEP_INTERVAL = 20 # in milliseconds
dr = 0              # drop rate
TARGET_P = 99       # percentile
while True:
  cntr = dataplane.readCounters()
  # Estimate percentile @ target SLO
  p = estimate_p(cntr)
  # e.g. below 99% for SLO@99-th
  if p < TARGET_P:
    dr = min(0.99, dr + STEP)
  else
    dr = max(0, dr - STEP)
  dataplane.apply_drop_rate(dr)
  sleep(SLEEP_INTERVAL)
```

Listing 11.1 – SVEN's control loop logic

11.3.3 Client and server applications

SVEN is completely transparent to the client and server applications as it is implemented entirely in the transport protocol. SVEN depends on the clients' ability to deal with request rejections and request classification for different drop rates. R2P2's API already requires clients to define an error callaback function that deals with early rejections. Also, R2P2's API allows clients to define different request policies. Those policies can be used to apply different request rates or mark requests as non-droppable if they are critical. The above classification is non-SVEN specific, it can be used by other policy enforcing mechanisms, and it is already supported by the existing R2P2 API.

SVEN's proactive drop mechanism guarantees that requests reaching the server will be executed, thus avoiding the need for an application-specific cancellation mechanism as in



(a) Fixed (b) Exponential (c) Bimodal

Figure 11.2 – 99-th Latency for different service time distributions with $\bar{S} = 10\mu s$ without SVEN. The red line shows the latency SLO@300µs. The vertical grey line shows the throughput that violates that SLO and it is shown with the same colour in Figure 11.3.

Zeta [151]. From the server perspective the transmission of `FEEDBACK` messages are internal to the R2P2 stack and the application should not take special care.

11.4 Evaluation

To evaluate the effectiveness of SVEN in transforming a generic low latency RPC service to a tail-tolerant system we implemented the above design on a Tofino [28] programmable switch, splitting the functionality between the P4 dataplane and a Python control plane. We run a series of synthetic microbenchmarks in which we control the service time distribution to investigate how SVEN performs across service times without application-specific configurations.

We used the open-source DPDK-based version of R2P2 [131], modified it as described in Section 8.3, and deployed it on a 16-core Xeon E5-2650 server connected with an Intel x520 10GbE NIC. Each core exposes its own Tx and Rx queues for a total of 16 independent workers. We ran the middlebox that implements the dataplane described in Figure 11.1 in a Barefoot Tofino v1 Edgecore Wedge100BF-32X.

In our experiments we use three different service time distributions with the same average service time of $\bar{S} = 10\mu s$: a fixed, an exponential, and a bimodal distribution in which 10% of the requests are 10 times slower than the rest, similar to the evaluation performed in R2P2 [202]. We consider an SLO at 300 µs for the 99-th percentile latency, and we use a random FIFO scheduling policy, meaning that incoming requests are randomly assigned to queues by the switch and are executed in order by each core. The scheduling policy is implemented by the *app-specific scheduling logic* of the middlebox in Figure 11.1 and it is orthogonal to the SVEN design. We chose this policy for simplicity. On the client side we use the Lancet [201] load generator.

In our first experiment we run the server without SVEN to understand the system behaviour across different loads and identify the load level that will violate the latency SLO for each service time distribution. Figure 10.5 shows the 99-th percentile latency as a function of the

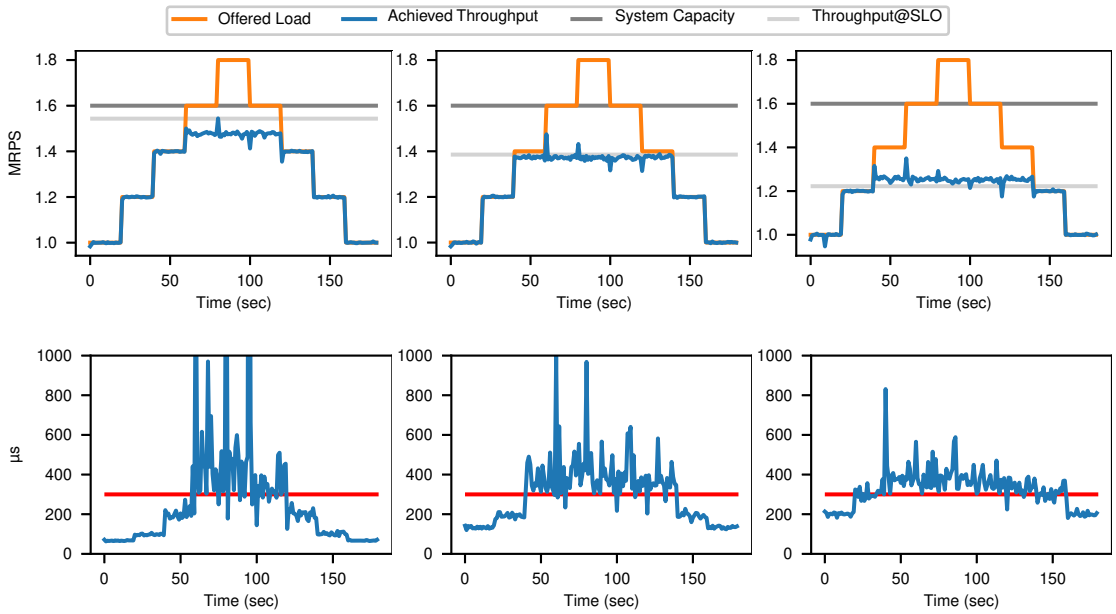


Figure 11.3 – Throughput (MRPS) and 99-th Latency (µs) over time for a step load pattern and the same 3 service time distributions. The difference between the orange and the blue throughput lines corresponds to SVEN's drop rate. The red line shows the latency SLO@300µs.

achieved throughput for the three service time distributions. We also plot the 99-th percentile latency SLO at 300 µs in red, and the throughput that violates the SLO in dashed grey. We observe that different service time distributions violate the SLO at different load levels: 1.54 MRPS for fixed, 1.39 MRPS for exponential, and 1.22 MRPS for bimodal. SVEN's purpose is to keep the offered throughput below those levels.

In the second experiment, we evaluate how SVEN performs as the offered request load changes. We change the offered load every 20 seconds starting from 1 MRPS up to 1.8 MRPS which is beyond the system capacity. Note that the system capacity is 1.6 MRPS — 16 cores and average service time of $\bar{S} = 10\mu s$. We measure the achieved throughput and latency every second and plot the results in Figure 11.3.

In the throughput plots we include the offered load, the achieved throughput, the system capacity, and the throughput that violates the SLO as identified in the previous experiment. The difference between the orange (offered load) and blue (achieved throughput) lines correspond to the drop rate as identified by the control plane. We observe that SVEN manages to approximately identify the load that will violate the latency SLO (light grey line) for each distribution and maintain throughput (blue line) close to that threshold even when the offered load was much higher and despite the simplicity of the control loop. We, also, see that SVEN does not waste throughput to achieve tail-tolerance. When SVEN drops requests the achieved throughput stays close to the throughput that violates the SLO.

The latency plots, in the same figure, show the 99-th percentile latency as a function of time and the latency SLO of 300µs for the same experiment. We observe that the tail latency stays close to the target SLO, even when the offered load is above the system capacity (when the orange line is above the dark grey line in the throughput graphs). In such cases of extremely high loads beyond capacity, latency would be arbitrarily high without SVEN.

The latency oscillation is explained by the AIAD control policy that requires a latency violation to control the drop rate. Note that different service times lead to different oscillations. This is explained by the slope of the latency versus throughput curve at the point of the SLO violation. A drop rate misconfiguration can lead to a more significant SLO violation in the case of the fixed service time distribution where the slope is steeper compared to the bimodal case. We expect that a more robust and carefully tuned control loop will reduce the observed latency oscillation.

11.5 Discussion and Future Work

Layering and Placement: The current SVEN implementation runs in the network on P4 switches, already used for RPC scheduling in R2P2, is placed before the scheduling logic, and servers multiple servers. However, our design is not limited to programmable switches. An alternative would be to implement SVEN's control and dataplane within servers. Such a deployment assumes the server is able to accurately estimate the time a request spends waiting and being processed. This would require the server to timestamp the first packet of a request once it enters the system before any queueing time and the last last reply packet. To do so, servers should depend either on hardware timestamping at the NIC or on an asymmetric design with a dispatcher, similar to RamCloud [281] or Shinjuku [178]. Note though, that this approach uses CPU resources that could be used for RPC serving. To avoid CPU usage, SVEN could also run on a smartNIC on the serverside. The smartNIC has full visibility to incoming requests and outgoing replies and can easily estimate the time a request spends on the server by tracking the request reception and the reply transmission time. A server-based SVEN should not be combined with an in-network scheduler though, since the scheduler would waste resources to schedule requests that are eventually dropped at the server.

SVEN, being an in-network solution, poses an interesting deployment question regarding its placement inside a complex RPC service. In our evaluation we only looked at a single-level application and placed SVEN before the RPC server. However, in a more complicated fan-out/fan-in application SVEN can be employed both for the leaf and the aggregator nodes. Although it depends on the application type and whether it can leverage the latency vs completeness trade-off, we suggest employing tail-tolerant mechanisms at the higher levels of the application, *e.g.*, the aggregator nodes, letting other mechanisms, *e.g.*, scheduling to deal with the leaf nodes. This way SVEN can better capture the user perceived latency, as opposed to the latency of a leaf service.

Control plane: SVEN currently depends on a very simplistic control plane and it treats equally all requests. As future work we would like to look at different alternatives for the control loop design and investigate their impact on latency oscillation. Also, future control loops should consider different request types and compute different drop rates for each request type, as those are defined by the R2P2 request type or policy. The control loop can depend on either more complicated heuristics [300], control theory [284], or online learning [132]. Given the separation between the control and the dataplane and the fact that the control loop only identifies the drop rate, thus not being on the critical path, the duration of the control loop could be in the order of 100s of µs. This can open up the design and implementation space for more accurate but slower implementations.

Dropped vs degraded requests: In our experiments we assumed that clients can leverage the latency vs completeness trade-off and did not re-issue the dropped requests. However, this will not always be the case. Depending on the deployment clients can have different options. First, we described that clients could prioritize requests or mark them as non-droppable based on existing R2P2 mechanisms. Also, in case of a replicated service clients can re-issue a dropped request to another server. The early rejection notification enables a more informed and less wasteful version of tied requests [80].

SVEN depends on explicit request drops to control latency in order to eliminate the need for server modification. Another approach that requires server support and cooperation, though, is one that marks requests as degraded instead of dropping them completely. So, as tail latency approaches the SLO SVEN changes the request type to `degraded-request` before forwarding them to the server based on the same probabilistic logic. A server would provide a reply of less quality back to the client, *e.g.,* an image with lower resolution, for a degraded request. Similar ideas have already been explored in previous works [121], and in different domains, *e.g.,* in the NDP congestion control scheme [143] that forwards only packet headers and drops the packet payload instead of dropping the entire packet.

Stricter Guarantees: SVEN is only the first step towards tail-tolerant systems and does not provide any strong guarantees at the moment, while it depends on the assumption that requests can be dropped. We believe that providing stronger tail-tolerance guarantees is viable but will require more complicated application analysis, such as such performance contracts [164]. Also, avoiding dropped or degraded requests can be achieved through redundancy similar to fault-tolerance. Combining performance verification, redundancy, and in-network control loops on top of the right abstractions can lead to by construction tail-tolerant systems.

11.6 Related Work

The idea of dynamic admission control has been explored in different contexts [357, 91, 364, 363]. These approaches were either application specific or targeting different deployment en-

vironments. SVEN is a transport layer mechanism, thus application agnostic, and can provide guarantees in µs-scale for datacenter applications based on in-network programmability. In the storage domain, Reflex [196] controls application latency through achieved throughput, while Mittos [145] leverages early rejections for fast request re-issue. Nebula [327] implements dynamic RPC admission control in hardware. Prior work that is closest to SVEN is our previously proposed SLO-aware flow control for TCP [197]. SVEN does not use complex queueing formulas to identify the right rate and depends on a better-fitted request abstraction.

11.7 Chapter Summary

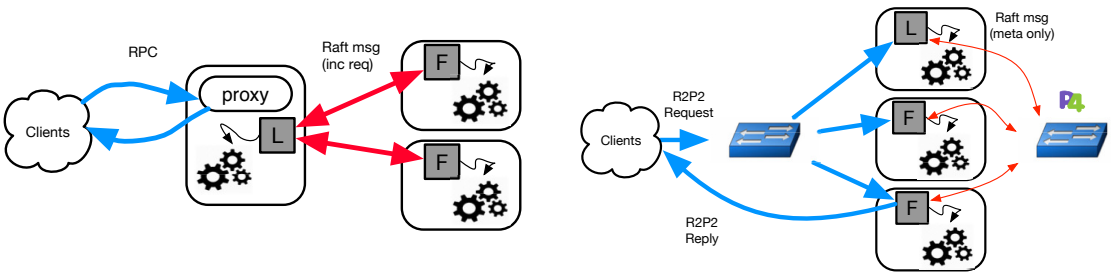
We advocate for tail-tolerance as a system design principle instead of a best-effort system metric. As a first step in this direction, we propose SVEN, an application-agnostic system for in-network SLO-aware RPC admission control implemented as part of the R2P2 transport protocol.

12 HovercRaft: Scalability and Fault-tolerance for µs-scale Services

12.1 Introduction

Warehouse-scale datacenters operate at an impressive degree of availability and information consistency despite constant component failures at all layers of the network, hardware and software stacks [29], constrained by the well-known theoretical tradeoffs between consistency, availability, and partition-tolerance [42]. This is the result of the careful decomposition of modern applications into components that each exhibit well-defined consistency, scalability, and availability guarantees. Each component is further configured to match specific service-level objectives, often expressed in terms of tail latency [80, 81].

Replication plays a key role in achieving these goals across the entire spectrum of tradeoffs. First, scalability is commonly achieved by managing replicas with relaxed consistency and ordering requirements [81, 42]. This is commonly deployed as a combination of caching layers, data replication, and data sharding [81]. Second, replication is also the foundation for fault-tolerance, whether achieved through fault-tolerant hardware and process pairs [137] or more commonly in datacenters through distributed consensus protocols of the Paxos and Raft families [276, 174, 274, 207, 208, 209, 210]. Such protocols ensure fault-tolerance through state



(a) Scalability bottlenecks of Raft.

(b) HovercRaft, with optional in-network acceleration using a P4 ASIC.

Figure 12.1 – Eliminating bottlenecks of SRM. Illustration on a 3-node cluster.

machine replication (SMR) [314], in which a distributed system with n nodes can offer both safety and liveness guarantees in the presence of up to f node failures as long as $n \geq 2 \times f + 1$ (under some network assumptions [117]).

Scalability and fault-tolerance are classically opposed, even though both rely on the same design principle of replication. Adding nodes can improve scalability with relaxed consistency and can lead to very large deployments within and across datacenters (*e.g.,* content delivery networks). On the flip side, adding nodes to a consensus system can improve fault tolerance but harm performance. In practice, most deployments of SMR are limited to small cluster sizes, *e.g.,* three or five replicas [71, 48, 157] as deployments of SMR with more than a handful of nodes reduce performance and are considered too expensive [36, 157].

12.1a shows the leader node bottlenecks for a classic SMR deployment using Raft: (1) the leader acts as the RPC server for all clients; (2) the leader must communicate individually with each follower to replicate messages and ensure their ordering. 12.1a also illustrates that a user-defined application that operates on the state machine must be modified to accept messages delivered by Raft (rather than use a more conventional RPC API transport).

In this work we focus on stateful datacenter applications that require fault-tolerance, low-latency, and scalability. We ask two main questions: (1) *Can we build fault-tolerant services in an application-agnostic, reusable manner, i.e., take an existing application (with deterministic behavior) and have it transparently utilize a SMR protocol?* (2) *Can we take advantage of the replication present for fault-tolerance to also improve the performance of the application?*

12.1b illustrates the key contributions of our system, HovercRaft, and its optional extension (HovercRaft++). We answer the first question primarily by integrating the Raft protocol [276] directly within R2P2 [202], a transport protocol specifically designed for in-network policy enforcement over remote procedure calls (RPC) inside a datacenter.

We answer the second question by first extending Raft to separate request replication from ordering and using IP multicast and in-network accelerators (*e.g.,* a P4 ASIC) to convert leader-to-multipoint interactions into point-to-point interactions. These enhancements reduce the performance degradation associated with larger clusters. We complete the answer to the second question by using R2P2's fundamental mechanism for in-network operations, which allows the destination IP address of an RPC request (as set by the client) to differ from the source IP of the reply. This allows the load balancing of client replies, as well as of the execution of read-only, totally-ordered requests. These changes reduce the I/O and CPU bottlenecks and allow HovercRaft to deliver even superior performance to an unreplicated (and therefore not fault-tolerant) deployment of the same application.

We make the following contributions:

- We integrate Raft [276], a widely used consensus algorithm, with R2P2 [202], a transport

protocol specifically designed for datacenter RPCs, to offer fault-tolerance transparently to applications.

- We propose HovercRaft, a set of Raft protocol extensions that leverage the built-in features of R2P2 to systematically eliminate I/O and CPU bottlenecks associated with SMR, without changing the core of the algorithm, thus its liveness and safety guarantees.

- We further take advantage of in-network accelerators now commonly found in datacenter switches to statelessly offload low-level message processing, thus eliminating scalability bottlenecks due to cluster size.

Our implementation of Raft relies on kernel-bypass to deliver up to 1M ordered operations per second in a series of microbenchmarks on a 3-node cluster, which corresponds to a 4× improvement over the state-of-the-art [103, 222, 293]. Our implementation of HovercRaft++ delivers 1M ordered operations for clusters of up to 9 nodes. The careful elimination of CPU and IO bottlenecks allows almost linear speedup over the unreplicated configuration for selected workloads. Our evaluation of Redis running YCSB-E shows that HovercRaft can deliver up to $142k$ YCSB-E operations per second on a 7-node cluster in ≤ 500µs at the 99th percentile, a 4× performance increase over the unreplicated case.

The HovercRaft codebase is opensource and can be found here [1].

12.2 Motivation

SMR Bottlenecks

We first focus on the normal case of operation for the Raft algorithm as described in Figure 2.1. We identify bottlenecks that might arise based on the interactions between the clients and the fault-tolerant group of servers, as well as the consensus communication pattern inside the fault-tolerant group.

Leader IO bottleneck for request replication: The leader is in charge of replicating requests to the followers. Once those requests start increasing in size or the number of followers increases, the transmission bandwidth of the leader's NIC will become the bottleneck, thus limiting throughput.

Leader IO bottleneck for client replies: The leader must also reply to all clients. Depending on the reply sizes, the leader transmission bandwidth can again become a bottleneck, especially in combination with the replication traffic.

[1] https://github.com/epfl-dcsl/hovercraft

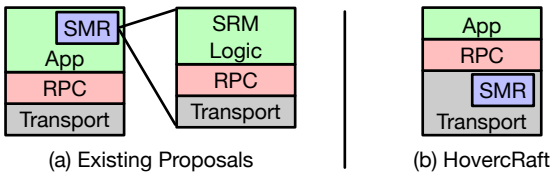


Figure 12.2 – HovercRaft proposal in terms of layering compared to existing approaches.

Leader CPU bottleneck for running requests: Since the leader is expected to reply to the client, it must run all client requests, even read-only ones. However, this can lead to a CPU bottleneck in the leader if the read-only operations are expensive. This, for example, can be the case in systems supporting mixed OLTP and OLAP workloads [302].

Leader packet processing rate: The leader must send requests and receive replies from the majority of the followers in order to make forward progress. Increasing quorum sizes increases the packet processing requirements at the leader. This can both limit the throughput of committed client requests if the IOPS becomes a bottleneck, and increase latency before hitting the IOPS bottleneck.

12.3 Design

Our goal is to provide a systematic and application-agnostic way of building fault-tolerant, µs-scale, datacenter services, with similar or better performance to the unreplicated ones. We achieve this by integrating the consensus directly within the RPC layer (Subsection 12.3.1) and through a set of extensions to the Raft consensus protocol that do not modify the core algorithm, but only go after its CPU and IO bottlenecks. We call the Raft version with the performance extensions HovercRaft. 12.1b summarizes our design, and in the rest of the section we analyze each design choice.

12.3.1 SMR-aware RPC layer

Clients interact with stateful servers through RPCs to change or query the internal server state, making RPCs the natural place to implement a systematic, application-agnostic mechanism for fault-tolerance. In a standard design of a fault-tolerant service, clients interact with the nodes of a cluster through a standard RPC library that is oblivious to SMR. For example, `etcd` uses gRPC [139] for this interaction. This protocol layering requires the SMR leader, which receives the initial client RPC, to decode it, re-encode it within the SMR protocol for insertion into the consensus log, and finally forward it to the followers. The selection of the endpoint for the client RPC can be done via a stateless load balancer which hides the internal IP addresses of the cluster but does not improve performance since all client requests have to go through the leader, *e.g.*, the `etcd gateway` [106].

We advocate, instead, to incorporate consensus directly within an RPC library or a transport protocol, which has RPC semantics (*e.g.,* gRPC [139] or R2P2 [202]) and to provide fault-tolerance at the RPC granularity. Specifically, the SMR layer becomes part of the RPC layer which forwards RPC requests to the application layer only after those requests have been totally ordered and committed by the leader. Doing so can transform any existing RPC service with deterministic behavior into a fault-tolerant one with no code modifications.

We chose to work with R2P2 [202] and incorporate Raft [276] within the RPC processing logic. We chose R2P2 as it is a transport protocol specifically designed for datacenter RPCs that targets in-network policy enforcement and decouples the initial request target from the replier. This design choice in R2P2 is crucial for RPC-level load balancing implemented at its core.

Figure 12.2 describes the proposed design in terms of layering. Unlike previous approaches that incorporate SRM libraries at the application layer, HovercRaft incorporates SMR within the same transport protocol that also exposes request-response pair semantics, making the solution application agnostic.

Our choice of Raft, instead of a Paxos variant, is driven by Raft's strong leader, and the in-order commit mechanism. In the original Paxos algorithm [207] there can be a different leader in each consensus round, thus limiting the potential for optimization due to centralized choice. Unlike Paxos, Raft is a consensus algorithm that depends on a strong leader in charge of ordering and replicating client requests across all followers, with a global view of all the nodes participating in the fault-tolerant group. Our goal is to take advantage of the global cluster view at the Raft leader in conjunction with R2P2's load balancing capabilities to go after the SMR bottlenecks. Although the above requirement is also satisfied by Multi-Paxos [208], Raft's in-order commit logic significantly simplifies the design of in-network acceleration for HovercRaft. We specifically target scalability bottlenecks, and we note that our design does not improve upon Raft's two network roundtrip approach, unlike 1-roundtrip proposals [222].

12.3.2 Separating RPC Replication from Ordering

The first Raft bottleneck of Section 12.2 is the leader IO transmission bottleneck due to request replication. With n nodes in the cluster, a mean RPC request size of S_{req}, and a link capacity of C, Raft cannot serve more than $C/((n-1)*S_{req})$ requests per second.

HovercRaft achieves fixed-cost SMR independent of the RPC request size. We solve the bottleneck by separating replication from ordering and leverage IP multicast to replicate the requests to all nodes. Instead of targeting a specific server, clients inside the datacenter send requests to a multicast group that includes the leader and the followers, or a middlebox that is in charge of assigning the correct multicast IP. All nodes in the group receive client requests without the leader having to send them individually to each of the follower nodes.

Obviously, IP-multicast does not guarantee ordering or delivery. As with Raft, the leader decides the order of client requests. R2P2 provides a way to uniquely identify an RPC based

on a 3-tuple *(req_id, src_port, src_ip)* and HovercRaft relies on this metadata built into the protocol. Upon reception of an RPC request, the leader immediately adds it to its log, while followers insert the RPC into a set of unordered requests. The leader then communicates fixed-size request metadata to the followers. Followers retrieve the RPC requests from the unordered set based on the request metadata and add them in their log. Therefore, in the common case, when no packet loss occurs, the leader is only in charge of ordering requests and not data replication.

12.3.3 Load balancing replies

The second bottleneck mentioned in Section 12.2 refers to the cost of replying to clients from the leader. We observe that replying to the client can be load balanced between the leader and the followers as long as the followers keep up with the request execution. By doing so, the I/O bottleneck due to client replies could expand from C/S_{reply} RPS to almost $n * C/Sreply$ RPS, where C is again the link capacity and S_{reply} is the reply size. For example, a server with a 10G NIC can serve up to approximately 400k requests per second, when the replies are 2 MTUs (we assume an MTU of 1500 bytes). Load balancing those replies among 3 replicas can improve application throughput by 3×. It also reduces the leader CPU load due to network protocol processing.

The leader can decide which node replies to the client for which request, given its global view of the cluster. According to the semantics of the vanilla Raft algorithm, a leader sends an `append_entries` message to the followers that includes the client requests and the leader commit index, so that followers can update their own commit index. There is no explicit commit message per log entry. As a result, the leader has to designate the replier when announcing the request order to the followers, and not after committing them.

HovercRaft extends the information stored in the Raft log with a `replier` field. The leader sets the `replier` field immutably for each entry before sending the particular entry to any follower for the first time. When a log entry is later committed, each Raft node can run the RPC, but only the one with the matching replier identifier replies to the client. For this, we use a built-in feature of R2P2 that allows the source IP address of the RPC reply to differ from the destination IP address of the request.

The load balancing of replies creates a window of uncertainty between the `append_entries` request that communicates the designated replier, and the commit point, during which the designated replier can fail, and as a result the client will not receive a reply. Note that introducing this window does not affect the correctness of the SMR algorithm, as it is consistent with Raft's semantics that do not guarantee exactly-once execution, and can lead to missed replies. Note also that clients may receive their replies in a different order than in the log; this is not, however, different than Raft.

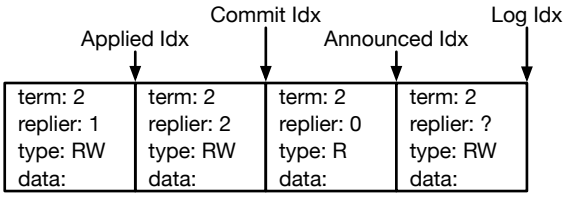


Figure 12.3 – A view of the leader log. Each log entry includes the term, the designated replier, a pointer to the client request, and whether it is read-only. The replier is set only for the entries up until the `announced_idx`.

12.3.4 Bounded Queues

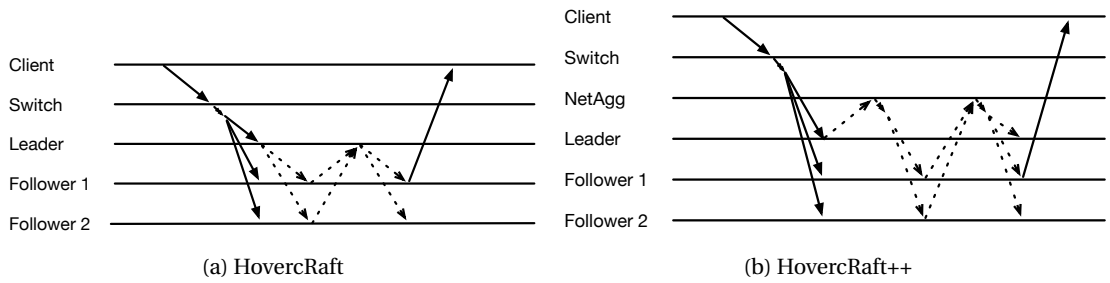
We rely on another design idea introduced in R2P2, *bounded queues*, to minimize the potential visible impact of a particular node failure to clients to at most B lost replies. R2P2's *Join-Bounded-Shortest-Queue* was designed as a load balancing policy to mitigate tail latency across stateless servers [202].

Join-Bounded-Shortest-Queue splits queueing between a large centralized queue and distributed bounded queues, one per server. The rational behind this policy is to delay delegating requests to a specific server queue in anticipation of better scheduling, approaching the performance of the optimal single queue. HovercRaft delays the reply node assignment to bound the number of lost replies in the case of node failure.

Specifically, HovercRaft caps the quantity of announced entries from the leader's log relatively to the applied index, thus bounding the amount of assigned but not applied operations. Figure 12.3 describes the different indices on the log: (1) the leader inserts entries at `log_idx`, the head of the log without determining yet which node will send the reply; (2) the leader selects the node in charge of replying to the client and updates the `announced_idx` accordingly; (3) the `commit_idx` represents the point upon which consensus has been reached; (4) the `applied_idx` represents the point upon which operations have been applied to the state machine. Each follower also has its own set of `applied_idx`, `commit_idx`, and `log_idx` indices on its local log. `Announced_idx` is relevant only for the leader. Followers communicate their `applied_idx` to the leader as part of the `append_entries` reply.

For every node there is a bounded queue of reply assignments to that node between its `applied_idx` and the leader's `announced_idx`. The leader respects the invariant of the bounded queue at node selection time, *i.e.,* when moving `announce_idx`: nodes with too many operations left to apply are not eligible for receiving additional work. This obviously includes the case when the node has failed and its `applied_idx` does not progress. Thus, choosing nodes based on the bounded queues minimizes the risk of selecting a failed node and eventually losing client replies.

We note that respecting the bounded queue invariant never affects liveness. When no node is eligible for designated replier, the leader simply waits either for the application in the leader



(a) HovercRaft (b) HovercRaft++

Figure 12.4 – The communication patterns in our design resulting from separating replication from ordering and using in-network fan-in/fan-out. Solid arrows refer to application requests and replies. Dotted arrows refer to SMR messages.

node to make progress and selects itself as a replier, or an `append_entries` reply from a follower that will make this follower eligible to reply.

12.3.5 Load balancing Read-only Operations

The third bottleneck of Section 12.2 is the leader CPU bottleneck. We observe that many RPCs only query the state machine but do not modify it. Such read-only requests still need to be placed in the Raft log and ordered to guarantee strong consistency, but do not need to be executed by all nodes. The load balancing design of Subsection 12.3.3 therefore naturally extends to the CPU for read-only operations, and can increase the global CPU capacity of the system. Clients tag their requests as read-only as part of the metadata of the R2P2 RPC. This information is also kept in the Raft log and propagated to the followers (see Figure 12.3). It follows that all requests remain totally ordered by Raft, but only the designated replier node executes a read-only query.

Read leases [136], which are an alternative solution for read-only requests, were initially proposed for Paxos [56] and also used in Chubby [48] and Spanner [71]. In this approach, read-only operations run on the leader without running consensus for them. However, this increases the CPU load and traffic on the leader. Megastore [25] grants leases to every replica for different read operations, but requires communicating with every replica to perform a write request. Quorum leases [257] also load balance read-only requests among different nodes, but assume an application-specific way of detecting read-write dependencies and do not match our application agnostic requirements. Finally, there is also the choice of not ordering read operations, acknowledging the risk of returning stale replies [104]

12.3.6 Join-Bounded-Shortest-Queue

Bounded queues are necessary to limit the lost client replies up to the queue bound, but can also help improve the end-to-end request latency, especially in cases where read-only requests

have high service time variability. Once the leader decides to announce more log entries, identifies the eligible followers, and has to select which follower will reply to the client. One option is random choice among eligible nodes. Another one is to leverage R2P2's *Join-Bounded **Shortest** Queue (JBSQ)* policy [202].

In HovercRaft, the leader maintains the queue depth of requests to be executed on each node. This counter is increased every time the leader assigns a request and decremented when followers reply to the leader. The JBSQ policy load balances the requests based on the known queue depths, as this is known to improve tail-latency compared to a random selection [202], by choosing the shortest among the bounded queues. The queue of a follower that is assigned to run a long read-only request will fill up while the node is busy serving the request and that will prevent the leader from assigning more work to that node. Preferring the other less loaded nodes is expected to improve the tail-latency in cases of high service-time variability, compared to the naive random assignment policy.

12.3.7 Communication in HovercRaft

12.4a summarizes the logic and the communication pattern described in the above design. A client sends requests to a pre-defined multicast group. Requests get replicated to all nodes based on multicast functionality existing in commodity switches. The leader orders requests and sends `append_entries` with request metadata to followers, identifying the request type (read-only or read-write) and delegating client replies; in this particular case follower 1 will reply. The followers reply to the leader and the leader commits the request. At the next `append_entries` request the followers are notified about the leader's current commit index, execute the client request, and follower 1 replies to the client. The above communication pattern increases latency in the unloaded case (2.5 RTTs), but can lead to significant throughput benefits.

12.4 HovercRaft++

The last bottleneck of § 12.2 is the packet processing overhead at the leader that becomes worse as the number of followers increases. The leader has to send `append_entries` requests to each follower independently and receive replies for each request. Communication delays inside the datacenter, though, are short and usually predictable, so it is likely that all followers make progress at the same pace. Based on that observation, we tackle the packet processing bottleneck using in-network programmability. Specifically, we design and implement an in-network aggregator, based on a P4-enabled programmable switch, that will handle the fan-out and fan-in of the `append_entries` requests and replies. This in-network accelerator should be viewed as part of the leader. A P4 switch runs the aggregation logic at line rate. So, we offload some of the leader's packet processing duties, thus reducing the leader's CPU pressure, on a hardware appliance specifically designed for packet IO. Our goal is to achieve fixed-cost SMR in the non-failure case independently of the number of followers for small

MSGs\System	Raft	HovercRaft	HovercRaft++
Rx msgs	1+(N-1)	1+(N-1)	1+1
Tx msgs	(N-1)+1	(N-1) + 1/N	1+1/N

Table 12.1 – Comparison of Rx and Tx message overheads for the leader in Raft, HovercRaft, and HovercRaft++ for the non-failure case. N is the number of nodes.

cluster sizes.

12.4b describes our proposed design. Similarly, to HovercRaft's communication pattern (12.4a), a client sends a request to the multicast group and this request gets replicated to all nodes. However, in the case of in-network acceleration (HovercRaft++), the leader, instead of communicating with each individual follower, sends only one `append_entries` request to the network aggregator. This request includes the metadata to implement the aforementioned load balancing logic. The network aggregator then forwards this request to the equivalent multicast group excluding the leader. The followers reply back to the aggregator and the aggregator keeps track of the per-follower replies, without forwarding them to the leader. Once the majority of the followers have successfully replied, the aggregator multicasts an `AGG_COMMIT` to all the nodes in the group, announcing the new commit index. Based on that message, the delegate follower (follower 1) replies back to the client.

Table 12.1 summarizes the communication complexity at the leader in Tx and Rx messages for the different approaches for a cluster with N nodes (N-1 followers and a leader). In the case of Raft, the leader receives the client request, sends N-1 `append_entries` requests to the followers, receives N-1 `append_entries` replies, and sends the reply to the client. In the case of HovercRaft, the leader receives the client request, sends N-1 `append_entries` requests (smaller than in the previous case) to the followers, receives N-1 `append_entries` replies, and approximately sends only 1/N replies to the client because of replies load balancing. Finally, in the case of HovercRaft++, the leader receives the client request, sends only one `append_entries` request to the aggregator that multicasts it to the followers, the aggregator collects the quorum and sends one `append_entries` reply to the leader. Similarly with the previous case, the leader approximately sends only 1/N replies to the client.

12.5 HovercRaft vs Raft

HovercRaft does not modify the core of the Raft algorithm but instead goes after its bottlenecks and implements optimizations to bypass them. Those optimizations are only in effect in the non-failure mode of operation. HovercRaft falls back to vanilla Raft whenever a failure, *e.g.,* failed leader, is detected. As a result, HovercRaft provides exactly the same *linearizability* guarantees as Raft. It assumes the same failure mode, and guarantees safety and liveness with $2f+1$ nodes, where up to f nodes can fail.

Raft's correctness depends heavily on the strong leader and its election process. HovercRaft's

modifications to the Raft logic only affect the normal operation after a leader is elected. As a result, the correctness of the leader election is not challenged.

The rest of this session discusses the modifications introduced by HovercRaft, how they affect the consensus logic, and how HovercRaft handles failures.

Separating replication and ordering: In HovercRaft, all nodes receive client requests through the multicast group and the leader is in charge of ordering. Client requests are placed in the Raft log as they are received only in the leader. Followers keep those requests in a list of un-ordered requests waiting for an `append_entries` request.

Followers index unordered requests based on R2P2 unique 3-tuple *(req_id, src_ip, src_port)*. Clients are responsible to ensure the uniqueness of the metadata identifiers. This is not a problem in practice given the large R2P2 metadata namespace. The leader can also include a hash of the request body to avoid cases of metadata collision. Therefore, there is a unique mapping between metadata in the *append_entries* message and the requests in the followers unordered list.

HovercRaft does not assume reliable multicast [153, 287]. Consequently, there might be cases in which client requests do not reach all the nodes. Followers detect such cases when processing an `append_entries` message and do not find the equivalent client request in their unordered set. We introduced a new `recovery_request` message type. Followers use this request to ask for a missing client request from the leader or any other follower that might have potentially received it. Once a follower retrieves a missing request, it adds it to the unordered set, waiting for the next `append_entries` request from the leader to order it properly.

The inverse case, in which the followers received a client request but the leader did not, does not require changes to the algorithm. The followers periodically garbage collect client requests in their unordered set that linger, based on a specific timeout. Early garbage collection does not affect the correctness of the algorithm and will unnecessarily trigger the recover mechanism described above.

Bounded Queues: Bounding the amount of committed but unapplied requests does not affect the number of lost client requests in case of a leader failure. Followers have also received the client requests already placed in the failed leader log, but not yet announced. When a leader fails, the new leader will remove the received client requests from its unordered set, add them to its log in some order and start sending `append_entries` announcing their order.

Load Balancing Client Replies: Raft does not guarantee exactly-once RPC semantics but instead only at-most-once RPC semantics [329]. It only guarantees linearizability of operations, leaving the client outside the algorithmic logic. Consequently, the client reply can be lost or the leader can crash after committing a log entry and before replying to the client. Guaranteeing exactly-once RPC semantics is outside the scope of Raft and projects like RIFL [216] solve

the problem of implementing exactly-once semantics on top of infrastructures providing at-most-once RPC semantics.

HovercRaft's ability to load balance replies introduces this window of uncertainty between the point of replier choice by the leader and the actual client reply. This is consistent with Raft's at-most-once RPC semantics, does not affect correctness, and should be considered equivalent to the cases of missing replies in vanilla Raft.

Load Balancing Read-Only Operations: Read-only operations do not modify the state machine but still need to be ordered to guarantee strong consistency. It is safe to execute them only in the designated replier, only after they have been committed. The application (client-side or server-side) is responsible to correctly identify which operations are read-only, as to avoid a catastrophic inconsistency.

In-network Aggregation: The aggregator should be viewed as part of the leader that undertakes leader tasks in the non-failure case. In HovercRaft++, followers reply to the aggregator only when `append_entries` requests succeed. When an `append_entries` fails, *e.g.,* due to wrong previous entry, followers talk directly to the leader, bypassing the aggregator. When the leader receives a failure reply for an `append_entries` request, it uses point-to-point communication with this follower until it recovers. In the meantime, this follower receives the multicast `append_entries` requests from the aggregator and keeps them in order to identify when to stop the recovery process with the leader, and continue with using the messages from the aggregator instead.

If the in-network aggregator fails, the followers stop receiving requests from the leader. This will trigger a new election process, in which the aggregator does not participate. Once a new leader is elected based on the vanilla Raft election process, the new leader has to identify whether to use the in-network aggregator or not. The leader switches to HovercRaft++ once it has confirmed the liveness of the in-network aggregator. The state in the aggregator is flushed after every new leader election.

Model-checking the correctness of HovercRaft++ using TLA+ [334] is left for future work.

12.6 Implementation

We implemented HovercRaft and HovercRaft++ based on the open-source version of R2P2 [131], and a production-grade, open-source implementation of Raft [49], which is thoroughly tested for correctness, and used in Intel's distributed object store [79]. We built the network aggregator in $P4_{14}$ and ran it in a Tofino ASIC [28].

To guarantee timely replies, we dedicate a thread to network processing and a thread to run the application logic. The networking thread is in charge of receiving client requests and

running the R2P2 and consensus logic. The application thread is in charge of running the application and replying to the client if necessary. In our implementation on top of DPDK [88], we configure 1 RX queue for the networking thread and 2 TX queues, one for each thread. The networking thread polls the RX queue, while the application thread polls for changes in the `commit_idx` and applies the newly committed entries.

12.6.1 R2P2 protocol extensions

We extended R2P2 to integrate consensus in its RPC processing logic. R2P2's RPC-tailored semantics and its RPC-aware design choice are a perfect fit to achieve our goal.

The R2P2 header includes two relevant fields for our implementation. The first one is the `POLICY` field, initially used to define load balancing policies. We extended the semantics of this field with two new policies. Clients use those fields to tag requests that must be totally ordered for strong consistency. Specifically, requests that read and modify the state machine should be marked with `REPLICATED_REQ`, and requests that only read with `REPLICATED_REQ_R`. Marking requests that require strong consistency allow servers in the fault-tolerance group to serve also other requests that are not replicated, with the probability of stale data, similarly to `etcd` [105]. Those non-replicated requests can also be load balanced based on the techniques described in [202].

The second relevant field is the message type. Given that Raft itself depends on RPCs, Raft RPCs are also on top of R2P2. We added two more message types, one for Raft requests and one for Raft responses, as to separate them from the client ones since they have to be handled by the consensus logic in R2P2. These fields are also used by the network aggregator to specially handle Raft requests and replies.

12.6.2 Raft extensions

We added minimal modifications to the Raft implementation initially for high throughput and low-latency and then to support HovercRaft. Specifically, we switched from the periodic application of the log, to eager application the moment the entries are committed. Then, we extended the log entry with two new fields to include the replier identifier and the entry's type (read-only/read-write). Finally, without modifying the Raft code, we extended the `append_entries` reply to include the applied index, necessary for the bounded queues (Subsection 12.3.4) and load balancing.

12.6.3 Multicast Flow Control and Recovery

Raft and HovercRaft differ noticeably in one particular area. In vanilla Raft, the leader is the only one receiving client requests and is in charge of both ordering and replicating them; it is the only bottleneck in the system. Therefore, dropping client requests at the leader is a form of

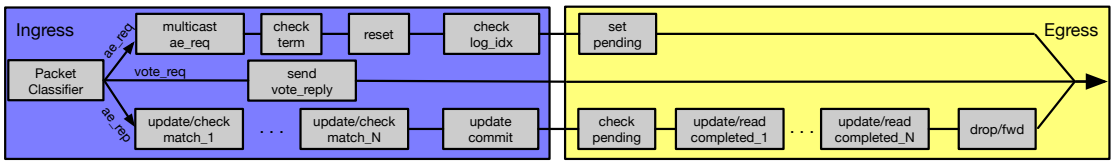


Figure 12.5 – The in-network aggregator pipeline handling `append_entries` requests and replies

implicit flow control. HovercRaft, however, leverages multicasting to replicate client requests, which implies that, under high load, different requests will be dropped for the leader and the followers.

As a consequence, HovercRaft has to implement flow control to guarantee forward progress under high load or bursts of load that lead to dropped multicast client messages. One way of dealing with the problem is to let the leader and the followers drop requests independently under high load, and rely on the recovery mechanism to create back-pressure. However, this would lead to poor performance.

Instead, we leverage the R2P2's `FEEDBACK` mechanism, that is designed to be repurposed according to the application needs, to limit the number of outstanding client requests in the system. For example, the R2P2 request router used `FEEDBACK` messages to implement the `JBSQ` scheduling policy. HovercRaft and HovercRaft++ use `FEEDBACK` messages to implement a coarse-grained flow control mechanism for multicast traffic.

Specifically, instead of letting clients send requests to a multicast IP, we use a middlebox, *e.g.*, a programmable switch, that counts the number of requests in the system and switches the destination IP to the multicast IP of the fault-tolerance group. Every time a node sends back a reply to the client, it also sends a `FEEDBACK` message to the flow-control middlebox, to decrement the counter of requests. When the number of requests in the system reaches a certain threshold, instead of multicasting, the middlebox sends a NACK back to the client for every new client request arriving, thus preventing throughput collapse in the system.

12.6.4 Aggregator implementation

Our in-network aggregator is implemented as part of a Tofino programmable switch, but it is an IP connected device that can be placed anywhere inside the data center. The aggregator maintains only soft state that is flushed on every new leader election.

The aggregator needs to keep per-follower state and the current commit index. For this purpose, we use P4 registers that can be read and modified in the dataplane. For each node the aggregator keeps its current log index and the number of completed requests necessary for load balancing. The current log index information in the aggregator is effectively the match index kept in the Raft leader, thus the aggregator should be seen as an extension of the leader

and not as a standalone entity. Our implementation uses two P4 registers for each follower, and each follower is handled in a different Tofino pipeline stage.

Figure 12.5 illustrates our implementation with the Tofino's ingress and egress pipelines. When receiving `append_entries` requests, the aggregator only has to forward the packet with a modified destination IP address set to the appropriate multicast group. That multicast group includes all nodes except for the sender. The aggregator keeps track of the current term. Receiving an `append_entries` request with a higher term will lead to the aggregator flushing its internal state.

Processing `append_entries` replies is more challenging since the aggregator needs to decide whether the log should be committed up until a certain point and multicast the `AGG_COMMIT` message to all the nodes. The `AGG_COMMIT` message should include the committed index and the number of completed requests per node. To achieve this in one pass through the dataplane, we keep the match_idx registers in the ingress pipeline and the completed_count registers in the egress pipeline. When an `append_entries` reply arrives at the aggregator, the aggregator updates the match index of the sender node, counts the number of nodes that have received up until this log index to determine whether it should commit or not, and sets its decision in per packet metadata. All replies go through egress processing, to at least update the register holding the completed requests. If the aggregator decides in ingress that the commit index is increased, it compiles an `AGG_COMMIT` reply that includes the completed requests of the followers and multicasts it to all nodes. Otherwise, it drops the reply in egress.

There are cases where the leader announces up to the same log index, which is already committed, in its `append_entries` request. This might happen either because there are no new client requests, or because a message between the aggregator and the leader was lost. In this case, the aggregator needs to forward the request to the followers to prevent a new leader election and send an `AGG_COMMIT` for an already committed log index. If the aggregator detects the same log index as in a previous `append_entries` request, it keeps track of it, and sends an `AGG_COMMIT` for the next `append_entries` reply it receives, even if the commit index is not increased. (`check_log_idx`, `set_pending` and `check_pending` stages)

The aggregator does not participate in the leader election, but it should be able to notify the new leader that it is up and ready to serve requests. Thus, the new leader, after being elected, contacts the aggregator sending a `vote_request` message. If the aggregator is up, it replies with a `vote_reply`. Note that this `vote_reply` does not count for the leader election.

Finally, Figure 12.5 illustrates the logic split between the ingress and egress pipelines required to meet the timing restriction of the ASIC: each stage of the pipeline can access only one register. With the Tofino v1 ASIC , HovercRaft++ can accommodate up to 9 nodes, with full line-rate processing, and without requiring any packet recirculation.

12.7 Evaluation

We evaluate HovercRaft and HovercRaft++ with the primary goal of showing the benefits of load balancing and in-network aggregation for datacenter SMR.

Our infrastructure is a of a mix of Xeon E5-2637 @ 3.5 GHz with 8 cores (16 hyperthreads), and Xeon E5-2650 @ 2.6 GHz with 16 cores (32 hyperthreads), connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. All machines are configured with Intel x520 10GbE NICs (82599EB chipset).

All experiments use the Lancet open-source load generator, which generates an open-loop Poisson arrival process, relies on hardware timestamping for accurate RPC measurements, and reports accurately the 99th percentile tail latency [201]. Lancet supports R2P2.

Our experiments compare four different system setups, all on top of DPDK:

(1) the unreplicated service (`UnRep`) is not fault-tolerant as the state-machine is not replicated. Clients interact with that single server using R2P2. This setup is expected to have the lowest latency, but it is not fault-tolerant.

(2) Our port of the vanilla Raft algorithm [49] on R2P2 and DPDK (`VanillaRaft`), which directly integrates the SMR layer within the RPC layer. This setup incorporates our design contributions from Subsection 12.3.1, but no protocol contributions.

(3) HovercRaft (`HovercRaft`), which incorporates protocol extensions to separate request replication from ordering (Subsection 12.3.2), and the ability to load-balance replies (Subsection 12.3.3) and read-only operations (Subsection 12.3.5).

(4) HovercRaft++ (`HovercRaft++`), which leverages in-network aggregation to offload protocol processing (Section 12.4). This last configuration leverages, in addition to the hardware above, a Barefoot Tofino ASIC that runs within an Edgecore Wedge100BF-32X accelerator connected to the Quanta switch via a 40Gbps link. We use the same Barefoot switch as a flow-control middlebox.

We use a combination of synthetic micro-benchmarks and a real-world application. Synthetic microbenchmarks depend on a synthetic service with configurable CPU service execution time, request, and reply sizes. Requests to this service can be either read-only or read-write. This methodology is used to determine protocol overheads in the presence of known upper bounds in terms of either CPU or I/O, and therefore to exercise the bottlenecks independently. For example, we run most experiments with a service time $S = 1\mu s$, which obviously limits the throughput to $\leq 1000KRPS$. Similarly, we run some experiments with 6KB replies, which limits throughput to $\sim\leq 200KRPS$ per 10GbE link.

The evaluation answers the following questions and quantifies the benefits of our design decisions:

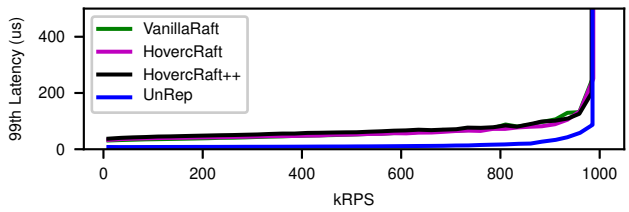


Figure 12.6 – Tail latency vs. throughput for a fixed service time $S = 1\mu s$ workload with 24-byte requests and 8-byte replies on N=3 node cluster.

1. what is the overhead of turning an RPC service into a fault-tolerant one with SMR implemented directly within the RPC layer in a 3-node cluster? (Subsection 12.7.1)
2. how is HovercRaft's and HovercRaft++'s performance affected by the client request size? (Subsection 12.7.1)
3. how do HovercRaft and HovercRaft++ scale with an increased number of nodes? (Subsection 12.7.2)
4. how does HovercRaft load balance replies and optimize read-only operations? (Subsection 12.7.3)
5. how does HovercRaft behave in the presence of failures? (Subsection 12.7.4)
6. how well does HovercRaft perform in practice with a production-grade application (Redis) and an industry-standard benchmark (YCSB-E)? (Subsection 12.7.5)

12.7.1 One million SMR operations per second

We first characterize all four setups on a 3-node cluster using a microbenchmark with a tiny service time ($S = 1\mu s$), minimum request size (24B) and minimum reply size (8B). There are no read-only operations to be load balanced. For this baseline experiment, we also explicitly disable the load balancing of client replies offered by HovercRaft, as to focus on protocol overheads.

Figure 12.6 shows the latency versus throughput curve for the four setups. We observe that there is a latency offset between the fault-tolerant configurations and the unreplicated case that comes from the extra round-trip required to achieve consensus. Nevertheless, that offset remains small and never exceeds 68µs, even for throughputs as high as $950KRPS$. Also, note that our experiment infrastructure depends on rather old hardware. Newer hardware, such as in [179], is expected to reduce this offset.

Figure 12.6 also shows that all four setups achieve close to the maximum possible throughput (1M RPS) under the 500µs SLO. This is a significant result as it outperforms other software-based, state-of-the-art approaches that either depend on kernel networking, such as NOPaxos [222]

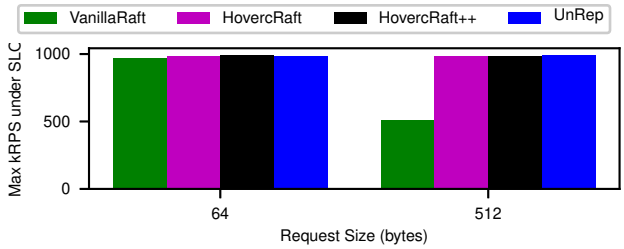


Figure 12.7 – Achieved throughput under a 500µs SLO for a fixed service time service workload with $S = 1\mu s$, 8-byte replies and different request sizes.

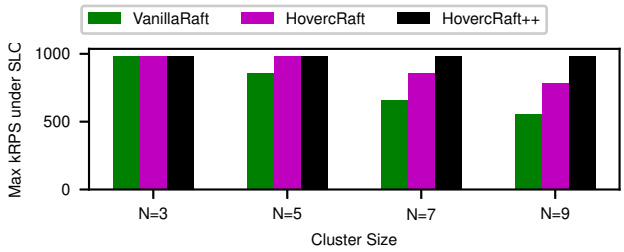


Figure 12.8 – Achieved throughput under a 500µs SLO for a workload with fixed service time of $S = 1\mu s$, 24-byte requests and 8-byte replies for different cluster sizes.

(by a factor of 4×), implement consensus inside the kernel, such as Kernel Paxos [103] (by a factor of 5×), or depend on RDMA [293] (by a factor of 4×). The difference is explained by our kernel-bypassed DPDK-based implementation and the leaner RPC protocol (R2P2) with its direct Raft integration.

Figure 12.7 adjusts the first experiment by setting the request size to 64B and 512B (compared to 24B in the first experiment) and reports the achieved client throughput in requests per second under the 500µs SLO. We observe that HovercRaft and HovercRaft++ are unaffected by the request size as they rely on multicast for request replication. However, the `VanillaRaft` configuration is sensitive to request size, with the throughput under SLO reduced by 2% and 48% for 64B and 512B sized-requests, respectively, vs 24B requests for the baseline experiment.

12.7.2 Scaling Cluster Sizes Without Regret

We now scale the cluster size to 5, 7, and 9 nodes *i.e.*, clusters that can tolerate 2, 3, and 4 failures, for the same experiment as the baseline in Subsection 12.7.1.

Figure 12.8 shows the achieved throughput under the 500µs SLO. In the 3-node cluster the three configurations are equivalent. The differences become obvious with larger clusters, with `VanillaRaft` most severely affected (−43% for 9 nodes). HovercRaft is unaffected up to 5 nodes, but shows a reduction with 7 and 9 nodes as the leader has to communicate independently with every follower. HovercRaft++ benefits from in-network aggregation and its performance is independent of the cluster size: the communication overhead at the leader is

always the same for any number of nodes, since the replication and aggregation are performed at the P4 switch, which operates at line rate.

12.7.3 Scaling to Improve Performance

In the previous experiments, we showed how HovercRaft++ outperforms the other configurations even without considering its load balancing benefits. We now enable the load balancing mechanism of HovercRaft with bounded queues of up to 128 pending requests and increase the reply size to 6KB; all other parameters remain the same as the baseline. All SMR operations execute on all nodes.

Figure 12.9 plots the latency as a function of the achieved throughput for the unreplicated case and HovercRaft++ with 3 and 5 nodes. As expected, the unreplicated setup hits an IO-bottlenecked at ~200$KRPS$, which corresponds to a fully utilized 10G link. Running on 3 and 5 nodes, increases the capacity of the system by almost 3× and 5×, since it is an IO-bottlenecked workload and all followers reply to clients.

We now study specifically the CPU load balancing mechanisms in our design. The purpose of this experiment is to study the impact of service time variability and scheduling disciplines on performance. For this, we assume that 75% of operations are read-only. We use the baseline configuration for request and reply sizes (which is free of I/O bottlenecks), and increase the CPU service time to an average of $S = 10\mu s$. We switch from the fixed service time distribution to a bimodal distribution, in which 10% of the requests are 10x longer than the rest. Based on these parameters, the unreplicated service is expected to reach close to 100k RPS, while HovercRaft++ on a 3-node cluster will be close to 200k RPS if perfect load balancing is achieved.

Figure 12.10 shows the 99th percentile tail-latency as a function of the achieved throughput for the unreplicated and replicated cases. For HovercRaft++, we consider two load balancing policies, `RANDOM` and `JBSQ`, with bounded queues of 32, due to the longer service time. We observe that load balancing the read-only operations increases the CPU capacity of the system for a 57% throughput improvement under the 500µs SLO. Also, the benefit of `JBSQ` over `RANDOM` becomes obvious. `JBSQ` allows HovercRaft++ to deliver lower latency, and therefore higher throughput under SLO, since it load balances read-only requests better by avoiding overloaded followers. We expect the gap between the 2 curves to increase with the number of nodes in the cluster given the more opportunities for careful load balancing.

12.7.4 Loadbalancing in the presence of failures

HovercRaft and HovercRaft++ enable pushing the achieved throughput load beyond the capacity of a single node by leveraging the existing redundancy. In the presence of failures, though, the capacity of the fault-tolerant system reduces. In such cases, HovercRaft and HovercRaft++ should manage the failure and gracefully degrade their performance.

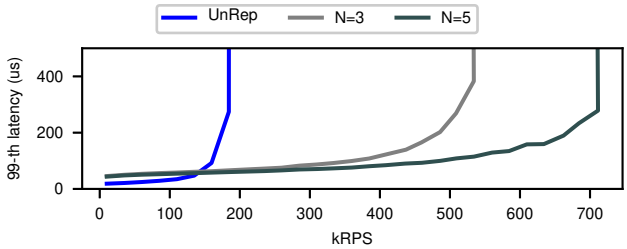


Figure 12.9 – Latency versus throughput for $S = 1\mu s$ fixed service time, 24-byte requests, and 6kB replies for different cluster sizes.

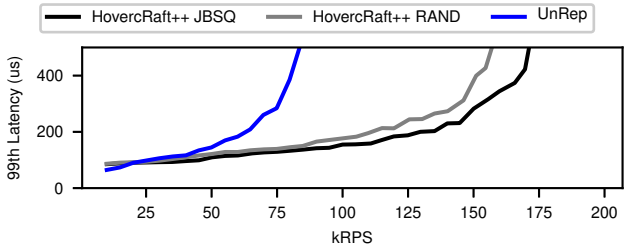


Figure 12.10 – Latency versus throughput for a $S = 10\mu s$ service time with bimodal distribution on a 3-node cluster. Requests are 24-bytes, replies are 8-bytes, and 75% of operations are read-only.

In the next experiment we study how HovercRaft++ behaves when the leader fails. We use the same setup as in the previous experiment with the bimodal distribution of $\bar{S} = 10\mu s$ and 75% read-only operations. Note that the capacity of the system with this request mix is 200k RPS with 3 nodes, but drops to 160k RPS with 2 nodes. We load the system with a fixed load of 165 kRPS which is below maximum capacity for the 3-node case, but above the maximum capacity for the 2-node setup. We configure flow control to allow up to 1000 client requests in the system. We measure the latency and throughput every second. At some point in time we kill the leader and we study how the system behaves.

Figure 12.11 plots the 99-th percentile latency and throughput as a function of time. We observe that before the leader failure the system serves 165k RPS under low latency. When the leader fails, one of the followers takes over and the system operates with 2 nodes. Because of bounded queues the new leader does not assign work to old leader. Throughput drops to the system capacity of 160k RPS. The flow control mechanism drops approximately 5k RPS maintaining the number of requests in the system below 1000, leading to increased latency, but avoiding collapse.

12.7.5 YCSB-E on Redis

Finally, we evaluate our design on a real-world application that requires generic SMR functionality for fault-tolerance. We run Redis [304] with the YCSB-E [70] workload.

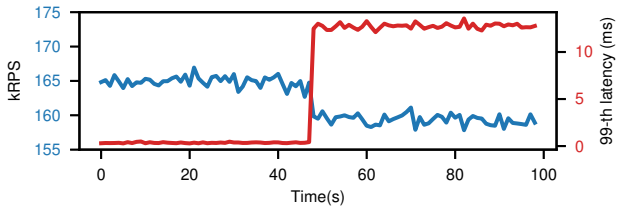


Figure 12.11 – 99-th percentile latency and throughput as a function of time in the presence of failures a leader failure for a HovercRaft++ 3-node cluster and a bimodal distribution of $\bar{S} = 10\mu s$ with 75% read-only operations.

YCSB-E is a cloud workload that consists of `SCAN` and `INSERT` operations, in a 95:5 ratio, modelling threaded conversations. It assumes 1kB records, with 10 fields of 100 bytes each. `INSERT` requests add a new record and they have to be ordered since they are parts of a conversation. `SCAN` requests query the last posts in a conversation, and they also need to be ordered for correctness, but they are read-only operations, thus they can be load balanced. We set the maximum number of elements to be returned in a `SCAN` request to 10.

Redis is an in-memory data store that supports basic data-structures and operations on them, such as lists, hashmaps, and sets. We chose Redis because it can be easily extended through user-defined modules [305]. Through Redis modules, users can define their own operations that manipulate Redis data-structures. We leverage the feature to implement the `SCAN` and `INSERT` operations of YCSB-E as single Redis operations that are guaranteed to execute within an isolated transaction. Given the support for arbitrary SMR operations, turning Redis to fault-tolerant requires an application agnostic approach similar to HovercRaft and HovercRaft++.

We ported Redis to use R2P2 instead of TCP for client operations. Beyond this protocol change, running the fault-tolerant version of Redis via `VanillaRaft`, HovercRaft, or HovercRaft++ required *no* code modifications, showing the benefits of transport layer support for SMR.

Figure 12.12 plots the 99th percentile latency as a function of the achieved throughput for the unreplicated case and the cluster configurations for HovercRaft++ with 3, 5, and 7 nodes. YCSB-E on Redis is a CPU-bound. read-mostly workload. We observe that SMR has only a very moderate negative impact on tail latency at low loads (up to $10KRPS$), but that HovercRaft++'s ability to leverage data replication present in SMR substantially increases the achieved throughput under the 500µs SLO. In the 7-node cluster, Redis can execute $142k$ YCSB-E operations per second under SLO, while guaranteeing full state machine replication and ordering of all operations, for a speedup of 4× over the unreplicated case. This speedup of 4× is consistent with the upper bound predicted by Amdahl's law given the relative cost of `SCAN` and `INSERT`, and the fact that only `SCAN`s can be load balanced.

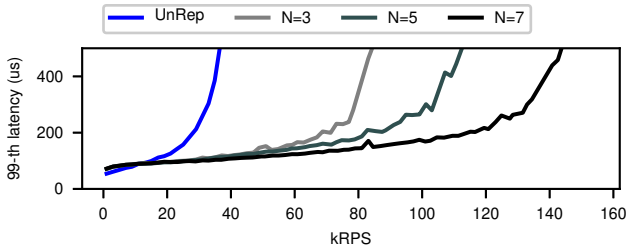


Figure 12.12 – Latency vs throughput for YCSB-E(95% SCAN 5% INSERT) on Redis with our custom module to support YCSBE-E operations.

12.8 Discussion

Programmable Switches and Consistency: HovercRaft++ is not the first system to leverage programmable switches for state machine replication. However, the use of programmable switches in HovercRaft++ differs when compared to the previous work, and we believe our proposal could be used in other distributed systems mechanisms, such as byzantine fault tolerance and primary backup.

We split the proposals of using programmable switches in SMR in three main categories. The first category includes systems that use P4 switches as **sequencers**. NOPaxos [222] and Harmonia [381] take advantage of in-network compute to assign sequence numbers to client requests under very low latency and high throughput. Despite its simplicity, one drawback in this approach is handling switch failures. After a sequencer failure, the new sequencer has to make sure that it respects the request order from the previous sequencer. Thus, these systems depend on a second fault-tolerant group at the level of the network controller that maintains an epoch number, that increases at every sequencer failure, and is used to bootstrap the new sequencer.

The second category includes systems that **fully offload** the implementation of an SMR algorithm on a programmable switch. For example, Paxos made switch-y [76] runs a Paxos coordinator and acceptor inside the P4 dataplane. Although such proposals can significantly improve performance, they suffer from the P4 dataplane limitations, such using fixed-size small values.

HovercRaft++ **partially offloads** leader duties to the programmable switch by using it as a very efficient packet processor. The programmable switch deals with Raft's fan-out and the fan-in communication patterns, while maintaining only soft state. If a switch fails, a new switch can take over starting from an empty state, thus bypassing the problems in the first category. Also, using a software-based leader running in a server offers a lot of flexibility to the system, unlike the proposals from the second category.

HovercRaft and High Speed Networks: HovercRaft and HovercRaft++ go after both IO and

CPU bottlenecks in SRM. Although the IO bottlenecks might become less important with the advent of faster networks, *e.g.,* 40G and 100G, the read-only operation load balancing and in-network fan-out/fan-in management employed in HovercRaft++ focus only on CPU bottlenecks and remain relevant despite bandwidth abundance. Especially, HovercRaft++ will become much more beneficial in those high speed networks since it offloads packet IO to the programmable switch and exposes a fixed overhead to the leader independent of the cluster size.

12.9 Related work

SMR: Consensus and state machine replication have been widely studied both from a theoretic [276, 174, 274, 207, 208, 209, 210], and a systems point of view. Systems such as Spanner [71], Zookeeper [157], Chubby [48], etcd [105] depend on those algorithms and are widely deployed serving millions of users. Researchers have optimized consensus systems to offer SMR in a WAN environment [256, 237], inside the datacenter [222, 294], implemented within the kernel [103], using RDMA fabrics [359, 293], or on top of FPGAs [162]. Similar to NOPaxos [222] and Speculative Paxos [294], HovercRaft focuses on fault tolerance inside the datacenter assuming lossy Ethernet fabrics (rather than RDMA). Despite leaderless approaches such as Mencious [237] and EPaxos [256] deal with leader bottlenecks, they lack a global cluster view, unlike HovercRaft, thus reducing the load balancing potentials.

Scaling read-only operations: Read leases [136] proposed to optimize for read-only operations in SMR and have been used either in the form of master leases in Spanner [71], and Chubby [48], or in the form of read quorums [257]. Those approaches either overload the leader or assume application-specific knowledge. HovercRaft implements load balancing of linearizable, read-only operations in an application-agnostic manner.

µs-scale computing: Exposing the hardware potential of µs-scale interactions within a datacenter to applications requires a new approach. This approach includes datacenter-specific operating systems [34, 290, 299, 377, 178, 280], user-level networking stacks [191, 167, 179], and transport protocols [202, 255]. HovercRaft builds on the R2P2 paradigm of pushing support for RPC to the transport layer [202] by extending it to offer fault-tolerance.

Networking protocol design and implementation: The separation of request data and metadata for ordering is also used in previous systems [37, 26, 113, 176]. UDP has been proposed to increase scalability of datacenter RPCs [271]. Bounded batching has been proposed to increase throughput [34] and reduce tail latency [202]. In-network programmability and programmable switches have been used for fault-tolerance, for sequencing in NOPaxos [222], to accelerate Vertical Paxos in NetChain [170], or implementing the entire Paxos algorithm [76, 78, 335]. Such approaches either do not assume switch failures, or rely on another fault-tolerant group

for state management. HovercRaft incorporates these ideas into the domain of SMR, and only stores soft-state in the network, which is recreated at every leader election.

Alternative fault-tolerance models: Fault-tolerance can be offered at the system call layer (Crane [73]), at the level of memory operations [77], in transactional databases [378, 221, 258], key-value stores [285], or for multi-threaded applications (Rex [141] and Eve [184]). HovercRaft provides fault-tolerance at the RPC layer in an application-agnostic manner and without code modifications.

12.10 Chapter Summary

We showed that replication can simultaneously improve both fault-tolerance and performance. Through the careful implementation of HovercRaft using modern kernel-bypass techniques and appropriate datacenter transport protocols, we first show that SMR is suitable for µs-scale computing, delivering 1 million ordered operations per second. Through the additional use of multicast, in-network accelerators, and load balancing, we tackle SMR's CPU and I/O bottlenecks and enable the deployment of fault-tolerant applications in an application-agnostic manner.

Wrapping Up Part IV

13 Future Directions

13.1 Latency vs Flow Completion Time

This thesis took a first step towards co-designing operating system and network mechanisms to improve µs-scale tail latency. However, the two communities (operating systems and networking) remain distinct and focus on different metrics and different bottlenecks. The networking community focuses on flow completion time and tries to minimize this through the extensive line of research on congestion control and packet scheduling. The operating systems community, on the other hand, focuses on end-to-end RPC latency and tries to optimise it through careful scheduling and system overhead elimination.

Given that flow completion time is part of the end-to-end RPC latency there is a significant potential in enhancing the design of congestion control and packet scheduling mechanisms with operating system and end-host metrics. We argue that blindly optimising for flow completion time using complicated congestion control algorithms does not make sense if the major part of end-to-end latency comes from queuing in the end-host and vice-versa. The common understanding in the network community is that *Shortest-Remain-Processing-Time* is the optimal policy for flow completion time [260] and recent state of the art approaches focus on that, *e.g.,* Homa [255]. A step closer to our proposed direction is coflow scheduling that schedules flows belonging to the same task and aims to optimize the total flow completion time [60, 61]. We believe that network and compute resources for RPCs can be modelled and scheduled in a similar way. A link can be viewed as a single-core CPU that can run to completion or with processor sharing, while a flow is a task. Thus, an RPC can be seen as a chain of three processors, the ingress link, the actual CPU, and the egress link. After modelling RPCs like this we can use the same queuing theory findings and system techniques to yield better end-to-end latency and tail-latency results, thus unifying existing scheduling and congestion control mechanisms along with the two research communities.

Another step the two communities have to make together is towards system support for ultra high speed (100G/400G) and reconfigurable networks. New hardware support is expected to entirely shift the existing bottlenecks. While 1-2 cores are enough to process minimum sized

packets on a 10G network, driving such high speed NICs will requires much more CPU power. The networking community is making steps towards this direction studying the impact of high speed [243] and reconfiguration [259] on existing networking mechanisms. The systems community should catch-up and try answering similar questions, *e.g.,* how to build an OS scheduler for an application running on an optical circuit-switched network that periodically schedules different flows. The goal should be to identify whether existing operating systems can remain oblivious to the network changes and how such networking design decisions should affect operating system design.

13.2 Looking Up the Stack

This thesis focuses on operating systems and network mechanisms without interfering with the programming language and the application logic itself. We believe that there is potential in designing applications specifically for predictable low latency latency. For example, state of the art latency-critical applications, such as MICA [229], focus on efficiency rather than predictability. An application that guarantees that any request type will finish in a predictable and capped amount of time would be much easier to schedule. Zygos [299] and Shinjuku [178] are complicated systems that tackle application-level unpredictability. IX [34], on the other hand, has a very simple design that leads to better performance but suffers when application service times vary. Designing applications assuming that the underlying operating system mechanisms are very efficient but simplistic can lead to an interesting separation of concerns. Figure 6.12 shows the potential of applications hinting operating system mechanisms and it is a step towards this direction.

Similarly, existing programming languages do not offer abstractions or mechanisms that can used to provide latency or scheduling guarantees. We believe there is work to be done in this domain. A first step towards this direction was Martin Weber's semester project [360]. In this work we extend Clang with automatic coroutine scheduling capabilities. Specifically, inspired by the Erlang scheduler [102] we created an LLVM pass that automatically yields execution to the coroutine scheduler by counting the number of executions in a loop and the function calls. Such a design is expected to remove the need for complex preemption-based operating system approaches, such as the ones introduced in Zygos [299] or Shinjuju [178]. We envision language support that through static analysis and dynamic instrumentation reduces service time variability and enhances scheduling, thus improving tail latency.

13.3 Distrusting the Datacenter

Throughout the thesis we make the assumption that the datacenter is a trusted environment in which there are no malicious components, *e.g.,* eavesdroppers. This is a fundamental argument behind R2P2's unencrypted communication or HovercRaft's non-byzantine agreement. Raising this assumption poses new challenges for latency-critical datacenter systems.

More and more companies deploy encrypted datacenter communication schemes [109] both for internal and external workloads. Encryption is a CPU-intensive task that has to be performed on the critical path. Extending RPC transport protocols, *e.g.,* R2P2 and eRPC [179], with support for encrypted RPCs without losing the existing performance benefits is challenging and might require significant redesign. Doing so on top of a 400G network will require several changes across all the software layers and the hardware infrastructure.

Similarly, datacenter operators start requiring support for low latency byzantine agreements. This needs stems either from new cloud products, *e.g.,* multiparty computation such as Microsoft's CCF [310], or due to non fail-stop bugs that are a subset of byzantine failures. We believe the techniques we developed in HovercRaft and the use of emerging programmable network hardware can improve the throughput and latency of byzantine agreements while keeping the same provably correct algorithms, *e.g.,* PBFT [52].

14 Conclusion

In this thesis we specifically focused on latency-critical µs-scale datacenter systems and proposed ways to better evaluate and reason about the performance of these systems, ways to optimise the existing abstractions while remaining backwards compatible, and we showed how introducing new abstractions can improve performance by taking advantage of new programmable hardware.

Initially, we introduced our system evaluation methodology on microbenchmarks and discrete event simulations that enables us to isolate system bottlenecks and quantify system overheads. Then, we proposed LANCET and SLOG that are tools for latency experiments that generate high throughput load and measure latency accurately based on a robust experiment methodology and hardware timestamping.

Having the necessary tools at our disposal for analysis and testing, we focused on existing abstractions, namely serving RPCs on top of TCP. We proposed ZYGOS, a work-conserving scheduler for µs-scale RPCs on a multicore system that imitates the behaviour of a single-queue system through light-weight work stealing. We then looked at replicated cloud services and we proposed CRAB, which is an L4 load balancer able to implement rich scheduling policies, while participating only in the connection establishment phase and remaining off the datapath till the flow completion. CRAB introduced a backwards compatible modification to TCP that enables connection redirection. During our work with TCP, we identified that its flow control mechanism, despite strong, is inefficient and agnostic to end to end latency, thus we proposed an SLO-aware flow control mechanism for TCP that minimizes the latency SLO violations.

In the last part of the thesis, we proposed R2P2, a new transport protocol for datacenter RPCs and in-network policy enforcement, driven by the mismatch between TCPs bytestream abstraction and the message-oriented nature of RPCs. R2P2 enabled us to offload server functionality to the network and leverage in-network programmability on P4 switches. We used the new abstraction to implement in-network RPC scheduling based on a novel scheduling policy JBSQ that can load balance RPCs across machines while achieving maximum throughput

and eliminating head of line blocking. We implemented a similar to TCP SLO-aware flow control mechanism as an R2P2 policy with SVEN. Finally, we showed how the new abstraction of request-response pairs allows pushing new functionality to the network. We introduced application-agnostic fault-tolerant µs-scale RPCs with HovercRaft. HovercRaft eliminates the bottlenecks of state machine replication through careful load balancing, multicasting, and in-network fan-out/fan-in management.

Bibliography

[1] Mutilate codebase (commit d65c6ef). https://github.com/leverich/mutilate.

[2] SolarFlare networking interfaces. https://www.solarflare.com/.

[3] Treadmill codebase (commit 1bf2082). https://github.com/facebook/treadmill.

[4] A10 Networks. https://www.a10networks.com/.

[5] A. Agache and C. Raiciu. Oh Flow, Are Thou Happy? TCP Sendbuffer Advertising for Make Benefit of Clouds and Tenants. In *Proceedings of the 7th workshop on Hot topics in Cloud Computing (HotCloud)*, 2015.

[6] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: distributed congestion-aware load balancing for datacenters. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 503–514, 2014.

[7] M. Alizadeh, A. G. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[8] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 253–266, 2012.

[9] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 435–446, 2013.

[10] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 185–198, 2013.

[11] J. T. Araújo, L. Saino, L. Buytenhek, and R. Landa. Balancing on the Edge: Transport Affinity without Network State. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 111–124, 2018.

[12] ARISTA. Arista 7368X4. https://www.arista.com/en/products/arista-7368x4-series-datasheet-quick-look.

[13] S. Arslan and N. McKeown. Switches Know the Exact Amount of Congestion. In *Proceedings of the 2019 Workshop on Buffer Sizing*, pages 10:1–10:6, 2019.

[14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[15] AWS Bare Metal Machines. https://aws.amazon.com/about-aws/whats-new/2019/02/introducing-five-new-amazon-ec2-bare-metal-instances/.

[16] AWS Lambda. https://aws.amazon.com/lambda/.

[17] Aws elastic load balancing. https://aws.amazon.com/elasticloadbalancing/.

[18] Netflix on AWS. https://aws.amazon.com/solutions/case-studies/netflix-case-study/.

[19] Microsoft azure cloud computing services. https://azure.microsoft.com.

[20] Azure accelerated networking. https://docs.microsoft.com/en-us/azure/virtual-network/create-vm-accelerated-networking-cli.

[21] Azure kernel patches for accelerated networking. https://github.com/microsoft/azure-linux-kernel/blob/master/4.4.116/README.

[22] Azure load balancer. https://docs.microsoft.com/en-us/azure/load-balancer.

[23] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 455–468, 2015.

[24] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.*, 7(3):181–192, 2013.

[25] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 223–234, 2011.

[26] M. Balakrishnan, K. Birman, and A. Phanishayee. PLATO: Predictive Latency-Aware Total Ordering. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 175–188, 2006.

[27] T. Barbette, C. Tang, H. Yao, D. Kostic, G. Q. M. Jr., P. Papadimitratos, and M. Chiesa. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 667–683, 2020.

[28] Barefoot Networks. Tofino product brief. https://barefootnetworks.com/products/brief-tofino/, 2018.

[29] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[30] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[31] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, 2017.

[32] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 335–348, 2012.

[33] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI)*, pages 49–65, 2014.

[34] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[35] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). *RFC*, 7540:1–96, 2015.

[36] C. E. B. Bezerra, L. L. Hoang, and F. Pedone. Strong Consistency at Scale. *IEEE Data Eng. Bull.*, 39(1):93–103, 2016.

[37] K. P. Birman, A. Schiper, and P. Stephenson. Lightweigt Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

[38] A. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[39] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *Computer Communication Review*, 44(3):87–95, 2014.

[40] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. RFC 2544 (Informational), Mar. 1999. Updated by RFCs 6201, 6815.

[41] A. Bremler-Barr, D. Hay, I. Moyal, and L. Schiff. Load balancing memcached traffic using software defined networking. In *Proceedings of the 2017 IFIP Networking Conference*, pages 1–9, 2017.

[42] E. A. Brewer. Pushing the CAP: Strategies for Consistency and Availability. *IEEE Computer*, 45(2):23–29, 2012.

[43] B. Briscoe and K. D. Schepper. Scaling TCP's Congestion Window for Small Round Trip Times. Technical Report TR-TUB8-2015-002, BT, Bell Labs, 06 2015.

[44] Broadcom. Broadcom Trident3. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series.

[45] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

[46] N. Budhiraja and K. Marzullo. Highly-Available Services Using the Primary-Backup Approach. In *Workshop on the Management of Replicated Data*, pages 47–50, 1992.

[47] B. Burns, B. Grant, D. Oppenheimer, E. A. Brewer, and J. Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[48] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)*, pages 335–350, 2006.

[49] C implementation of raft. https://github.com/willemt/raft.

[50] 10 thousand connections problem. http://www.kegel.com/c10k.html, 1999.

[51] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, 2011.

[52] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation (OSDI)*, pages 173–186, 1999.

[53] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 7:1–7:13, 2016.

[54] Cavium. Cavium XPliant Ethernet Switch-Product Family. https://www.sdxcentral.com/articles/news/arista-brocade-using-cavium-xpliant-chips/2016/12/.

[55] Linux cfs scheduler. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[56] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, 2007.

[57] C.-C. Chang, S.-R. Yang, E.-H. Yeh, P. Lin, and J.-Y. Jeng. A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning. In *Proceedings of the 2017 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2017.

[58] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 21–28, 2005.

[59] I. Cho, K. Jang, and D. Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 239–252, 2017.

[60] M. Chowdhury and I. Stoica. Coflow: a networking abstraction for cluster applications. In *Proceedings of The 11st ACM Workshop on Hot Topics in Networks (HotNets-XI)*, pages 31–36, 2012.

[61] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with Varys. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 443–454, 2014.

[62] E. Cidon, S. Choi, S. Katti, and N. McKeown. AppSwitch: Application-layer Load Balancing within a Software Switch. In *Proceedings of the 1st Asia-Pacific Workshop on Networking (APNet)*, pages 64–70, 2017.

[63] Cisco. Cisco Nexus 9300-GX. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/datasheet-c78-741560.html.

[64] Cisco Systems. T-Rex: Cisco's realistic traffic generator. https://trex-tgn.cisco.com.

[65] Citrix Netscaler ADC. https://www.citrix.com/products/netscaler-adc/.

[66] D. D. Clark. The design philosophy of the DARPA internet protocols. In *Proceedings of the ACM SIGCOMM 1988 Conference*, pages 106–114, 1988.

[67] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Trans. Comput. Syst.*, 32(4):10:1–10:47, 2015.

[68] CloudSuite Benchmarking Suite. http://cloudsuite.ch/.

[69] G. Contreras and M. Martonosi. Characterizing and improving the performance of Intel Threading Building Blocks. In *Proceedings of the 2008 IEEE International Symposium on Workload Characterization (IISWC)*, pages 57–66, 2008.

[70] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SOCC)*, pages 143–154, 2010.

[71] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, 2013.

[72] The Network Impact of the Global COVID-19 Pandemic. https://thenewstack.io/the-network-impact-of-the-global-covid-19-pandemic/.

[73] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 105–120, 2015.

[74] A. Daglis, M. Sutherland, and B. Falsafi. RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, pages 35–48, 2019.

[75] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 373–387, 2018.

[76] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos Made Switch-y. *Computer Communication Review*, 46(2):18–24, 2016.

[77] H. T. Dang, J. Hofmann, Y. Liu, M. Radi, D. Vucinic, R. Soulé, and F. Pedone. Consensus for Non-volatile Main Memory. In *Proceedings of the 26th IEEE International Conference on Network Protocols (ICNP)*, pages 406–411, 2018.

[78] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: consensus at network speed. In *Proceedings of the Symposium on SDN Research (SOSR)*, pages 5:1–5:7, 2015.

[79] Daos object store. https://github.com/daos-stack.

[80] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[81] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[82] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Job-aware Scheduling in Eagle: Divide and Stick to Your Probes. In *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)*, pages 497–509, 2016.

[83] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 499–510, 2015.

[84] C. Delimitrou and C. Kozyrakis. Quality-of-Service-Aware Scheduling in Heterogeneous Data centers with Paragon. *IEEE Micro*, 34(3):17–30, 2014.

[85] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, pages 127–144, 2014.

[86] D. Dickey and W. A. Fuller. Likelihood ratio statistics for autoregressive time series with a unit root. *Econometrica*, 49(4):1057–72, 1981.

[87] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (SC)*, 2009.

[88] Data plane development kit. http://www.dpdk.org/.

[89] ANS (Accelerated Network Stack) on DPDK. https://github.com/ansyun/dpdk-ans.

[90] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

[91] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation (OSDI)*, pages 261–275, 1996.

[92] F. Duchene and O. Bonaventure. Making multipath TCP friendlier to load balancers and anycast. In *Proceedings of the 25th IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, 2017.

[93] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose schedular. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 261–276, 1999.

[94] A. Dunkels. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science*, 2:77, 2001.

[95] J. Eidson and K. Lee. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*.

[96] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 523–535, 2016.

[97] Number of daily exchanged emails. https://www.statista.com/statistics/456500/daily-number-of-e-mails-worldwide/.

[98] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Proceedings of the 15th ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 275–287, 2015.

[99] Envoy Reverse Proxy. https://www.envoyproxy.io/.

[100] Epollexclusive kernel patch. https://lwn.net/Articles/667087/, 2015.

[101] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 345–362, 2019.

[102] Erlang Scheduler Details and Why It Matters. https://hamidreza-s.github.io/erlang/scheduling/real-time/preemptive/migration/2016/02/09/erlang-scheduler-details.html.

[103] E. G. Esposito, P. R. Coelho, and F. Pedone. Kernel Paxos. In *Proceedings of the 37th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 231–240, 2018.

[104] etcd ordering guarantees. https://github.com/etcd-io/etcd/blob/master/Documentation/learning/api_guarantees.md.

[105] `etcd`: Distributed reliable key-value store for the most critical data of a distributed system. https://github.com/etcd-io/etcd.

[106] `etcd` Documentation Guide: What is etcd gateway? https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/gateway.md.

[107] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/-subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[108] F5 Networks, INC. https://f5.com/.

[109] Building Facebook's service encryption infrastructure. https://engineering.fb.com/security/service-encryption/.

[110] Number of Facebook users. https://www.statista.com/statistics/346167/facebook-global-dau/.

[111] Fibre channel protocol. https://en.wikipedia.org/wiki/Fibre_Channel_Protocol.

[112] Intel corp. intel 82599 10 gbe controller datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf.

[113] P. Felber and A. Schiper. Optimistic Active Replication. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 333–341, 2001.

[114] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII)*, pages 37–48, 2012.

[115] D. Firestone. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 315–328, 2017.

[116] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 51–66, 2018.

[117] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[118] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, and C. Paasch. TCP Extensions for Multipath Operation with Multiple Addresses. *RFC*, 8684:1–68, 2020.

[119] B. Ford. Structured streams: a new transport abstraction. In *Proceedings of the ACM SIGCOMM 2007 Conference*, pages 361–372, 2007.

[120] A. Fox and E. A. Brewer. Harvest, Yield and Scalable Tolerant Systems. In *Proceedings of The 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 174–178, 1999.

[121] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 78–91, 1997.

[122] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, 2011.

[123] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, pages 3–18, 2019.

[124] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: cloud scale load balancing with hardware and software. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 27–38, 2014.

[125] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. pHost: distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 2015 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 1:1–1:12, 2015.

[126] F. Gaud, S. Geneves, R. Lachaize, B. Lepers, F. Mottet, G. Muller, and V. Quéma. Efficient Workstealing for Multicore Event-Driven Systems. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 516–525, 2010.

[127] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 57–76, 2007.

[128] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, 2003.

[129] M. Ghobadi, R. Mahajan, A. Phanishayee, N. R. Devanur, J. Kulkarni, G. Ranade, P.-A. Blanche, H. Rastegarfar, M. Glick, and D. C. Kilper. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 216–229, 2016.

[130] S. Ghorbani, Z. Yang, P. B. Godfrey, Y. Ganjali, and A. Firoozshahian. DRILL: Micro Load Balancing for Low-latency Data Center Networks. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 225–238, 2017.

[131] R2P2 source code. https://github.com/epfl-dcsl/r2p2.

[132] R. Glaubius, T. Tidwell, C. D. Gill, and W. D. Smart. Real-Time Scheduling via Reinforcement Learning. pages 201–209, 2010.

[133] A. Goel, C. Krasic, and J. Walpole. Low-latency adaptive streaming over tcp. *ACM Trans. Multim. Comput. Commun. Appl.*, 4(3):20:1–20:20, 2008.

[134] Number of daily Google queries. https://www.internetlivestats.com/google-search-statistics/.

[135] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. ELI: bare-metal performance for I/O virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII)*, pages 411–422, 2012.

[136] C. G. Gray and D. R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–210, 1989.

[137] J. Gray. Fault Tolerance in Tandem Systems. In *Proceedings of the 1985 International Workshop on High-Performance Transaction Systems (HTPS)*, 1985.

[138] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2015.

[139] gRPC. http://www.grpc.io/.

[140] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 202–215, 2016.

[141] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: replication at the speed of multi-core. In *Proceedings of the 2014 EuroSys Conference*, pages 11:1–11:14, 2014.

[142] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 135–148, 2012.

[143] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 29–42, 2017.

[144] N. Hanford, V. Ahuja, M. Balman, M. K. Farrens, D. Ghosal, E. Pouyoul, and B. Tierney. Characterizing the impact of end-system affinities on the end-to-end performance of high-speed flows. In *Proceedings of the 3rd International Workshop on Network-Aware Data Management (NDM@SC)*, pages 1:1–1:10, 2013.

[145] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 168–183, 2017.

[146] HAProxy. https://haproxy.org.

[147] Haproxy dsr. https://www.haproxy.com/blog/layer-4-load-balancing-direct-server-return-mode/.

[148] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, pages 161–175, 2015.

[149] M. Harchol-Balter, C. Li, T. Osogami, A. Scheller-Wolf, and M. S. Squillante. Analysis of Task Assignment with Cycle Stealing under Central Queue. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 628–, 2003.

[150] K. He, E. Rozner, K. Agarwal, W. Felter, J. B. Carter, and A. Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 465–478, 2015.

[151] Y. He, S. Elnikety, J. R. Larus, and C. Yan. Zeta: scheduling interactive services with partial execution. In *Proceedings of the 2012 ACM Symposium on Cloud Computing (SOCC)*, page 12, 2012.

[152] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[153] H. W. Holbrook, S. K. Singhal, and D. R. Cheriton. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. In *Proceedings of the ACM SIGCOMM 1995 Conference*, pages 328–341, 1995.

[154] C. E. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. *RFC*, 2992:1–8, 2000.

[155] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. F. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the 21st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 271–282, 2015.

[156] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *Proceedings of The 18th ACM Workshop on Hot Topics in Networks (HotNets-XVIII)*, pages 60–68, 2019.

[157] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[158] Intel Corporation. Intel 82580EB/82580DB Gigabit Ethernet Controller Datasheet. http://www.intel.com/content/www/us/en/embedded/products/networking/82580-eb-db-gbe-controller-datasheet.html. Revision: 2.7, September 2015.

[159] Intel Corporation. Intel Ethernet Controller 710 Series Datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf. Revision: 2.9, April 2017.

[160] Number of active internet users. https://www.statista.com/statistics/617136/digital-population-worldwide/.

[161] S. Islam and M. Welzl. Start me up: Determining and sharing tcp's initial congestion window. In *Proceedings of the 2016 Applied Networking Research Workshop*, ANRW '16, pages 52–54, New York, NY, USA, 2016. ACM.

[162] Z. István, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 425–438, 2016.

[163] Ixia. Ixia traffic generator. https://www.ixiacom.com.

[164] R. R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. J. Argyraki, and G. Candea. Performance Contracts for Software Network Functions. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 517–530, 2019.

[165] V. Jacobson, R. T. Braden, and D. A. Borman. TCP Extensions for High Performance. *RFC*, 1323:1–37, 1992.

[166] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 113–129, 2017.

[167] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 489–502, 2014.

[168] N. D. E. Jerger, T. Krishna, and L.-S. Peh. *On-Chip Networks, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.

[169] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. F. Lofstead, K. Mohror, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. The Popper Convention: Making Reproducible Systems Evaluation Practical. In *Proceedings of the 31st IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1561–1570, 2017.

[170] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, 2018.

[171] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 121–136, 2017.

[172] J. Johnson. *GUI Bloopers 2.0: Common User Interface Design Don'ts and Dos*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2007.

[173] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[174] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 245–256, 2011.

[175] S. A. Jyothi, M. Dong, and B. Godfrey. Towards a flexible data center fabric with source routing. In *Proceedings of the Symposium on SDN Research (SOSR)*, pages 10:1–10:8, 2015.

[176] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 222–230, 1991.

[177] M. Kablan, A. Alsudais, E. Keller, and F. Le. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 97–112, 2017.

[178] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 345–360, 2019.

[179] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16, 2019.

[180] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 295–306, 2014.

[181] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 185–201, 2016.

[182] S. Kanev, K. M. Hazelwood, G.-Y. Wei, and D. M. Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *Proceedings of the 2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 31–40, 2014.

[183] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: predictable low latency for data center applications. In *Proceedings of the 2012 ACM Symposium on Cloud Computing (SOCC)*, page 9, 2012.

[184] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 237–250, 2012.

[185] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 485–497, 2015.

[186] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th ACM Symposium on the Theory of Computing (STOC)*, pages 654–663, 1997.

[187] H. Kasture and D. Sánchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, pages 729–742, 2014.

[188] H. Kasture and D. Sánchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 3–12, 2016.

[189] Facebook's katran. https://github.com/facebookincubator/katran.

[190] N. P. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research (SOSR)*, page 10, 2016.

[191] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. E. Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the 2019 EuroSys Conference*, pages 24:1–24:16, 2019.

[192] Kernel connection multiplexer. https://lwn.net/Articles/657999/, 2015.

[193] Kernel connection multiplexer patch. https://lwn.net/Articles/657970/, 2015.

[194] J. F. C. Kingman. The single server queue in heavy traffic. *Mathematical Proceedings of the Cambridge Philosophical Society*, 57(4):902–904, 1961.

[195] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[196] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*, pages 345–359, 2017.

[197] M. Kogias and E. Bugnion. Flow control for Latency-Critical RPCs. In *Proceedings of the 2018 Workshop on Kernel-Bypass Networks (KBNETS@SIGCOMM)*, pages 15–21, 2018.

[198] M. Kogias and E. Bugnion. HovercRaft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the 2020 EuroSys Conference*, pages 25:1–25:17, 2020.

[199] M. Kogias and E. Bugnion. Tail-tolerance as a systems principle not a metric. In *4th Asia-Pacific Workshop on Networking*, APNet '20, page 16–22, New York, NY, USA, 2020. Association for Computing Machinery.

[200] M. Kogias, R. Iyer, and E. Bugnion. Bypassing the Load Balancer Without Regrets. In *Proceedings of the 2020 ACM Symposium on Cloud Computing (SOCC)*, 2020.

[201] M. Kogias, S. Mallon, and E. Bugnion. Lancet: A self-correcting Latency Measuring Tool. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 881–896, 2019.

[202] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019.

[203] M. Kogias, M. Weber, and E. Bugnion. Slog: Your switch is also your load-generator. http://infoscience.epfl.ch/record/276585, 2020.

[204] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). *RFC*, 4340:1–129, 2006.

[205] Kubernestes Container Orechestrator. https://kubernetes.io/.

[206] O. Laadan, J. Nieh, and N. Viennot. Structured linux kernel projects for teaching operating systems concepts. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 287–292, 2011.

[207] L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[208] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[209] L. Lamport. Fast Paxos. *Distributed Comput.*, 19(2):79–103, 2006.

[210] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 312–313, 2009.

[211] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. R. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 183–196, 2017.

[212] Latency Is Everywhere And It Costs You Sales - How To Crush It. http://highscalability.com/blog/2009/7/25/latency-is-everywhere-and-it-costs-you-sales-how-to-crush-it.html.

[213] The Cost of Latency. https://perspectives.mvdirona.com/2009/10/the-cost-of-latency/.

[214] J.-Y. Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, Lausanne, Switzerland, 2010.

[215] C. Lee, C. Park, K. Jang, S. B. Moon, and D. Han. Accurate Latency-based Congestion Feedback for Datacenters. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 403–415, 2015.

[216] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. K. Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 71–86, 2015.

[217] A. Lerner, R. Hussein, and P. Cudré-Mauroux. The Case for Network Accelerated Query Processing. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[218] K.-C. Leung, V. O. K. Li, and D. Yang. An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges. *IEEE Trans. Parallel Distrib. Syst.*, 18(4):522–535, 2007.

[219] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the 2014 EuroSys Conference*, pages 4:1–4:14, 2014.

[220] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, and K. S. McKinley. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 14:1–14:13, 2016.

[221] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 104–120, 2017.

[222] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 467–483, 2016.

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

[224] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 31–44, 2016.

[225] Y. Li, D. Wei, X. Chen, Z. Song, R. Wu, Y. Li, X. Jin, and W. Xu. DumbNet: a smart data center network fabric with dumb switches. In *Proceedings of the 2018 EuroSys Conference*, pages 9:1–9:13, 2018.

[226] libevent. http://libevent.org/.

[227] libuinet a library version of freebsd's tcp/ip stack. https://github.com/pkelsey/libuinet.

[228] libuv. http://libuv.org/.

[229] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[230] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 301–312, 2014.

[231] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 450–462, 2015.

[232] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. G. Greenberg. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Perform. Evaluation*, 68(11):1056–1071, 2011.

[233] Apache lucene. https://lucene.apache.org/.

[234] Lucene nightly benchmarks. https://home.apache.org/~mikemccand/lucenebench.

[235] S. Mallon. tcp: Fix sof_timestamping_rx_hardware to use the latest timestamp during tcp coalescing. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=cadf9df27e7cf40e390e060a1c71bb86ecde798b, 2018.

[236] R. Mandeville and J. Perser. Benchmarking Methodology for LAN Switching Devices. RFC 2889 (Informational), Aug. 2000.

[237] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machine for WANs. In *Proceedings of the 8th Symposium on Operating System Design and Implementation (OSDI)*, pages 369–384, 2008.

[238] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, R. Ricci, and A. Klimovic. Taming Performance Variability. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, pages 409–425, 2018.

[239] I. Marinos, R. N. M. Watson, and M. Handley. Network stack specialization for performance. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 175–186, 2014.

[240] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. E. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413, 2019.

[241] Marvell. Marvell LiquidIO. https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics/liquidio-ii-smart-nics.html.

[242] J. M. McCauley, A. Panda, A. Krishnamurthy, and S. Shenker. Thoughts on load distribution and the role of programmable switches. *Computer Communication Review*, 49(1):18–23, 2019.

[243] R. McGuinness and G. Porter. Evaluating the performance of software NICs for 100-gb/s datacenter traffic control. In *Proceedings of the 2018 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 74–88, 2018.

[244] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 319–330, 2011.

[245] D. Meisner, J. Wu, and T. F. Wenisch. BigHouse: A simulation infrastructure for data center systems. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–45, 2012.

[246] Mellanox. Mellanox Innova2 SmartNIC. https://www.mellanox.com/products/smartnics/innova-2-flex.

[247] Mellanox Corporation. ConnectX-4 NIC. https://www.mellanox.com/products/infiniband-adapters/connectx-4.

[248] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter. RotorNet: A Scalable, Low-complexity, Optical Datacenter Network. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 267–280, 2017.

[249] Memaslap Load Generator. https://libmemcached.org/libMemcached.html.

[250] Memcached. https://memcached.org/.

[251] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 15–28, 2017.

[252] R. Mittal, V. T. Lam, N. Dukkipati, E. R. Blem, H. M. G. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 537–550, 2015.

[253] M. Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, 2001.

[254] In-memory mongodb. https://docs.mongodb.com/manual/core/inmemory/.

[255] B. Montazeri, Y. Li, M. Alizadeh, and J. K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 221–235, 2018.

[256] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in Egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

[257] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos Quorum Leases: Fast Reads Without Sacrificing Writes. In *Proceedings of the 2014 ACM Symposium on Cloud Computing (SOCC)*, pages 22:1–22:13, 2014.

[258] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 517–532, 2016.

[259] M. K. Mukerjee, C. Canel, W. Wang, D. Kim, S. Seshan, and A. C. Snoeren. Adapting TCP for Reconfigurable Datacenter Networks. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 651–666, 2020.

[260] A. Mushtaq, R. Mittal, J. M. McCauley, M. Alizadeh, S. Ratnasamy, and S. Shenker. Datacenter congestion control: identifying what is essential and making it practical. *Computer Communication Review*, 49(3):32–38, 2019.

[261] Mutilate Load Generator. https://github.com/leverich/mutilate.

[262] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 17–33, 2017.

[263] A. Narayan, F. Cangialosi, P. Goyal, S. Narayana, M. Alizadeh, and H. Balakrishnan. The Case for Moving Congestion Control Out of the Datapath. In *Proceedings of The 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*, pages 101–107, 2017.

[264] Netperf. http://www.netperf.org/netperf/.

[265] Netronome. Agilio CX SmartNICs. https://www.netronome.com/products/agilio-cx/.

[266] Nginx. https://www.nginx.com/.

[267] Nginx dsr. https://www.nginx.com/blog/ip-transparency-direct-server-return-nginx-plus-transparent-p

[268] Nginx thread pool usage. https://www.nginx.com/blog/thread-pools-boost-performance-9x/.

[269] NGINX Reverse Proxy. https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/.

[270] J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[271] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[272] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)*, pages 182–195, 2016.

[273] Open FastPath. http://www.openfastpath.org/.

[274] B. M. Oki and B. Liskov. Viewstamped Replication: A General Primary Copy. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 8–17, 1988.

[275] V. A. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless Datacenter Load-balancing with Beamer. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 125–139, 2018.

[276] D. Ongaro and J. K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, pages 305–319, 2014.

[277] Open OnLoad. http://www.openonload.org/.

[278] Open Compute. https://www.opencompute.org/.

[279] W. H. Organization et al. Coronavirus disease 2019 (covid-19): situation report, 72. 2020.

[280] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 361–378, 2019.

[281] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[282] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[283] The P4 Language Specification. https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf. Accessed on 20.09.2018.

[284] S. S. Parekh, N. Gandhi, J. L. Hellerstein, D. M. Tilbury, T. S. Jayram, and J. P. Bigus. Using Control Theory to Achieve Service Level Objectives In Performance Management. In *Integrated Network Management*, pages 841–854, 2001.

[285] S. J. Park and J. K. Ousterhout. Exploiting Commutativity For Practical Fast Replication. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 47–64, 2019.

[286] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. G. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 207–218, 2013.

[287] S. Paul, K. K. Sabnani, J. C.-H. Lin, and S. Bhattacharyya. Reliable Multicast Transport Protocol (RMTP). *IEEE J. Sel. Areas Commun.*, 15(3):407–421, 1997.

[288] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: a centralized "zero-queue" datacenter network. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 307–318, 2014.

[289] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 2012 EuroSys Conference*, pages 337–350, 2012.

[290] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. E. Anderson, and T. Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, 2016.

[291] Pf_ring. http://www.ntop.org/products/packet-capture/pf_ring/.

[292] B. Pit-Claudel, Y. Desmouceaux, P. Pfister, M. Townsley, and T. H. Clausen. Stateless Load-Aware Load Balancing in P4. In *Proceedings of the 26th IEEE International Conference on Network Protocols (ICNP)*, pages 418–423, 2018.

[293] M. Poke and T. Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 107–118, 2015.

[294] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 43–57, 2015.

[295] J. Postel. Transmission Control Protocol. *RFC*, 793:1–91, 1981.

[296] R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI)*, pages 293–306, 2010.

[297] K. Prasopoulos. Extending r2p2 with congestion control and request-level scheduling. http://infoscience.epfl.ch/record/277844, 2020.

[298] G. Prekas, A. Belay, M. Primorac, A. Klimovic, S. Grossman, M. Kogias, B. Gütermann, C. Kozyrakis, and E. Bugnion. IX Open-source version 1.0 – Deployment and Evaluation Guide. Technical report, EPFL Technical Report 218568, 2016.

[299] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[300] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC)*, pages 342–355, 2015.

[301] M. Primorac, E. Bugnion, and K. J. Argyraki. How to Measure the Killer Microsecond. In *Proceedings of the 2017 Workshop on Kernel-Bypass Networks (KBNETS@SIGCOMM)*, pages 37–42, 2017.

[302] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB)*, pages 36–45, 2013.

[303] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proceedings of the 2016 EuroSys Conference*, pages 36:1–36:15, 2016.

[304] Redis. https://redis.io/.

[305] Redis modules. https://redis.io/topics/modules-intro.

[306] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 101–112, 2012.

[307] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 123–137, 2015.

[308] Microsoft corp. receive side scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx.

[309] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *Proceedings of The 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2011.

[310] M. Russinovich, E. Ashton, C. Avanessians, M. Castro, A. Chamayou, S. Clebsch, M. Costa, C. Fournet, M. Kerner, S. Krishna, J. Maffre, T. Moscibroda, K. Nayak, O. Ohrimenko, F. Schuster, R. Schwartz, A. Shamis, O. Vrousgou, and C. M. Wintersteiger. Ccf: A framework for building confidential verifiable replicated services. Technical Report MSR-TR-2019-16, Microsoft, April 2019.

[311] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[312] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of The 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*, pages 150–156, 2017.

[313] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. *CoRR*, abs/1903.06701, 2019.

[314] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[315] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[316] Seastar Project. http://www.seastar-project.org/networking/.

[317] M. Shahbaz, L. Suresh, J. Rexford, N. Feamster, O. Rottenstreich, and M. Hira. Elmo: source routed multicast for public clouds. In *Proceedings of the ACM SIGCOMM 2019 Conference*, pages 458–471, 2019.

[318] Number of daily shipments. https://earlymoves.com/2017/03/21/amazon-3-million-packages-a-day-alibaba-12-million-packages-a-day/.

[319] Silo: Multicore in-memory storage engine. https://github.com/stephentu/silo.

[320] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 183–197, 2015.

[321] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI)*, pages 33–46, 2010.

[322] Speed Matters. https://ai.googleblog.com/2009/06/speed-matters.html.

[323] Spirent Communications. Spirent test modules and chassis. https://www.spirent.com/Products/TestCenter/Platforms/Modules.

[324] R. R. Stewart. Stream Control Transmission Protocol. *RFC*, 4960:1–152, 2007.

[325] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large DataBases (VLDB)*, pages 1150–1160, 2007.

[326] P. L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 513–527, 2015.

[327] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. N. Pnevmatikatos, and A. Daglis. The NEBULA RPC-Optimized Architecture. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, pages 199–212, 2020.

[328] Docker Swarm. https://docs.docker.com/engine/swarm/.

[329] B. H. Tay and A. L. Ananda. A Survey of Remote Procedure Calls. *Operating Systems Review*, 24(3):68–79, 1990.

[330] D. Thaler and C. E. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. *RFC*, 2991:1–9, 2000.

[331] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing Network Protocols at User Level. In *Proceedings of the ACM SIGCOMM 1993 Conference*, pages 64–73, 1993.

[332] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. I. T. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: a software-defined storage architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 182–196, 2013.

[333] Apache thrift. https://thrift.apache.org/.

[334] The tla+ homepage. https://lamport.azurewebsites.net/tla/tla.html.

[335] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman. The Case For In-Network Computing On Demand. In *Proceedings of the 2019 EuroSys Conference*, pages 21:1–21:16, 2019.

[336] TPC-C Benchmark. http://www.tpc.org/tpcc/, 2010.

[337] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, 2013.

[338] J. Turner. New directions in communications(or which way to the information age?). *IEEE communications Magazine*, 24(10):8–15, 1986.

[339] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, 2005.

[340] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Comput. Networks*, 53(11):1830–1845, 2009.

[341] Berkeley Internet Name Domain (BIND). https://www.bind9.net.

[342] Conntrack. https://manpages.debian.org/testing/conntrack/conntrack.8.en.html.

[343] dnsmasq. http://www.thekelleys.org.uk/dnsmasq/doc.html.

[344] extended Berkeley Packet Filter. https://www.iovisor.org/technology/ebpf.

[345] Intel Ethernet Flow Director - Application Targeting Routing. https://software.intel.com/content/www/us/en/develop/articles/setting-up-intel-ethernet-flow-director.html.

[346] Kubernetes Service. https://cloud.google.com/kubernetes-engine/docs/concepts/service.

[347] Netfilter. https://www.netfilter.org.

[348] Netperf. https://fossies.org/linux/netperf/doc/netperf.pdf.

[349] Enabling dsr on nginx. https://www.nginx.com/blog/ip-transparency-direct-server-return-nginx-plus-transparent-proxy.

[350] The QUIC Transport Protocol - IETF. https://tools.ietf.org/html/draft-ietf-quic-transport-27.

[351] Receive Flow Steering. https://lwn.net/Articles/382428.

[352] The Toeplitz Hash Algorithm. https://en.wikipedia.org/wiki/Toeplitz_Hash_Algorithm.

[353] eXpress Data Path. https://www.iovisor.org/technology/xdp.

[354] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 407–420, 2017.

[355] Varnish. https://varnish-cache.org/.

[356] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 2015 EuroSys Conference*, pages 18:1–18:17, 2015.

[357] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *USENIX Annual Technical Conference*, pages 189–202, 2001.

[358] Voltdb. https://www.voltdb.com/.

[359] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: fast and scalable paxos on RDMA. In *Proceedings of the 2017 ACM Symposium on Cloud Computing (SOCC)*, pages 94–107, 2017.

[360] Introducing scheduling decisions in cooperative multitasking: an automated approach. https://marioskogias.github.io/students/weber.pdf.

[361] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–104, 2015.

[362] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)*, pages 307–320, 2006.

[363] M. Welsh and D. E. Culler. Overload management as a fundamental service design primitive. In *ACM SIGOPS European Workshop*, pages 63–69, 2002.

[364] M. Welsh and D. E. Culler. Adaptive Overload Control for Busy Internet Servers. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[365] WhatsApp 2M Connections. https://https://blog.whatsapp.com/196/1-million-is-so-2011.

[366] A. Wierman and B. Zwart. Is Tail-Optimal Scheduling Possible? *Oper. Res.*, 60(5):1249–1257, 2012.

[367] The english wikipedia page article dump. https://dumps.wikimedia.org/enwiki/20180401.

[368] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-Szendy. Zygaria: Storage Performance as a Managed Resource. In *Proceedings of the 2006 Real-Time and Embedded Technology and Applications Symposium*, pages 125–134, 2006.

[369] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 543–557, 2015.

[370] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 309–322, 2016.

[371] K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 43–56, 2016.

[372] YCSB Load Generator. https://github.com/brianfrankcooper/YCSB.

[373] Number of YouTube video hours uploaded per minute. https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/.

[374] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

[375] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazières, and M. F. Kaashoek. Multiprocessor Support for Event-Driven Programs. In *USENIX Annual Technical Conference*, pages 239–252, 2003.

[376] H. Zhang, M. Dong, and H. Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologie (FAST)*, pages 167–180, 2016.

[377] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam. I'm Not Dead Yet!: The Role of the Operating System in a Kernel-Bypass Era. In *Proceedings of The 17th Workshop on Hot Topics in Operating Systems (HotOS-XVII)*, pages 73–80, 2019.

[378] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 263–278, 2015.

[379] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 456–468, 2016.

[380] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. WCMP: weighted cost multipathing for improved fairness in data centers. In *Proceedings of the 2014 EuroSys Conference*, pages 5:1–5:14, 2014.

[381] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin. Harmonia: Near-Linear Scalability for Replicated Storage with In-Network Conflict Detection. *Proc. VLDB Endow.*, 13(3):376–389, 2019.

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

[383] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 523–536, 2015.

[384] Zygos kernel. https://github.com/ix-project/zygos, 2017.

Marios Kogias

54 Chemin de la Prairie
Lausanne, 1007 Switzerland
+41788393244
marios.kogias@epfl.ch
https://marioskogias.github.io/

RESEARCH FOCUS

Operating Systems, Networking, Datacenters, Tail Latency, Remote Procedure Calls, Dataplanes, Scheduling, Load-balancing, Network Transports, Programmable Switches

EDUCATION

Doctor of Philosophy — September 2015 - August 2020
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
Advisor: Edouard Bugnion

Diploma (5-year degree, Master's equivalent) — October 2009 - October 2014
School of Electrical and Computer Engineering
National Technical University of Athens (NTUA), Greece
Thesis Advisor: Nektarios Koziris
GPA: 9.36/10.00 (top 2%)

EXPERIENCE

Research Intern — June 2018 - September 2018
Microsoft Research, Redmond, US
I worked as part of the Cloud Efficiency group on latency-critical VM scheduling.
Advisors: Ricardo Bianchini, Sameh Elnikety

Software Engineering Intern — June 2016 - September 2016
Google, Mountain View, US
I worked as part of the Platforms team on performance analysis of the Google datacenter network.
Advisor: Malweeka Tewari

Software Engineering Intern — November 2014 - April 2015
CERN, Geneva, CH
I worked on the invenio project, a digital library software, implementing a new search engine based on ElasticSearch.

Software Engineering Intern — May 2014 - August 2014
Google Summer of Code - LTTng project
I worked on implementing a tracing pipeline for Ceph's RADOS on LTTng and Twitter's Zipkin. This work was taken over by Ceph and now is part of their upstream repository.

Junior Developer — July 2013 - September 2013
okeanos, GRNET, Athens, GR
I contributed to the Synnefo project, specifically on the virtual networking infrastructure implementing the Openstack Neutron API for Synnefo in Python Django

Founder, Software Engineer — December 2010 - September 2013
sourcelair.com, Athens, GR
I worked on sandboxing untrusted code execution and the sourcelair iOS application.

CONFERENCE PUBLICATIONS

Bypassing the Load Balancer Without Regrets
Marios Kogias, Rishabh Iyer, Edouard Bugnion
SoCC 2020

HovercRaft: Scalability and Fault-tolerance for μs-scale Datacenter Services
Marios Kogias, Edouard Bugnion
Eurosys 2020

R2P2: Making RPCs first-class datacenter citizens
Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, Edouard Bugnion
Usenix ATC 2019

Lancet: A self-correcting Latency Measuring Tool
Marios Kogias, Stephen Mallon, Edouard Bugnion
Usenix ATC 2019

ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks
George Prekas*, Marios Kogias*, Edouard Bugnion (*equal co-authors)
SOSP 2017

WORKSHOP PUBLICATIONS

Tail-tolerance as a Systems Principle not a Metric
Marios Kogias, Edouard Bugnion
APNet 2020

Flow Control for Latency-Critical RPCs
Marios Kogias, Edouard Bugnion
KBnets@SIGCOMM18

POSTERS

SLOG: Your switch is also your load-generator
Marios Kogias, Martin Weber, Edouard Bugnion
NSDI 2020

Measuring Latency: Am I doing right?
Marios Kogias, Christos Kozyrakis, Edouard Bugnion
NSDI 2017

TEACHING

Principles of Computer Systems (EPFL, CS-522)	Winter 2017,2018,2019
Introduction to Operating Systems (EPFL, CS-323)	Spring 2019
Database Systems (EPFL, CS422)	Spring 2018
Mise a Niveau (EPFL, MAN)	Spring 2017
Computer Networks (EPFL, COM-208)	Winter 2016

PROFESSIONAL SERVICE

ASPLOS 2020 Artifact Evaluation Committee
Eurosys 2020 Shadow PC
Eurosys 2017 Shadow PC
External Reviewer for IEEE/ACM Transactions on Networking

AWARDS

Eurosys 2020 Best Student Paper Award
2019 IBM PhD Fellowship
EPFL PhD Fellowship 2015-2016

GRANTS Microsoft Swiss JRC, Project: TTL-MSR (co-PI)
Usenix Student Grant to attend NSDI 2017