IMPROVING SOFTWARE MIDDLEBOXES AND DATACENTER TASK SCHEDULERS

Hugo de Freitas Siqueira Sadok Menna Barreto

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientadores: Miguel Elias Mitre Campista
Luís Henrique Maciel Kosmalski Costa

Rio de Janeiro
Outubro de 2018

IMPROVING SOFTWARE MIDDLEBOXES AND DATACENTER TASK SCHEDULERS

Hugo de Freitas Siqueira Sadok Menna Barreto

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

Prof. Miguel Elias Mitre Campista, D.Sc.

Prof. Otto Carlos Muniz Bandeira Duarte, Dr.Ing.

Prof. Artur Ziviani, Dr.

Prof. Ítalo Fernando Scotá Cunha, Dr.

RIO DE JANEIRO, RJ – BRASIL
OUTUBRO DE 2018

Barreto, Hugo de Freitas Siqueira Sadok Menna

Improving Software Middleboxes and Datacenter Task Schedulers/Hugo de Freitas Siqueira Sadok Menna Barreto. – Rio de Janeiro: UFRJ/COPPE, 2018.

XV, 69 p.: il.; 29, 7cm.

Orientadores: Miguel Elias Mitre Campista

Luís Henrique Maciel Kosmalski Costa

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2018.

Referências Bibliográficas: p. 59 – 69.

1. Middleboxes. 2. Task Schedulers. 3. Fairness. I. Campista, Miguel Elias Mitre *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

To my parents and grandmother.

Agradecimentos

Muitas pessoas contribuíram para esta dissertação de forma direta ou indireta. A seguir há uma tentativa de agradecer a elas.

Primeiro agradeço aos meus pais Marcelo e Márcia Sadok, e à minha vó Carmen Siqueira, por sempre acreditarem em mim e por me darem suporte incondicional. Sem eles nada disso seria possível. Agradeço também aos meus irmãos Bruno e Luna Sadok. Bruno por aturar minhas piadas inoportunas, pelas excelentes conversas e por até mesmo revisar alguns textos (chatos segundo ele). Luna por jogar comigo e pelos desenhos mais legais que já recebi.

Esta dissertação também não existiria se não fossem os meus orientadores Miguel Campista e Luís Costa. Eles me orientaram desde a graduação e me deram liberdade para trabalhar nos problemas que eu mais gostava—por mais ecléticos que fossem. Sou muito grato por terem me introduzido à pesquisa em redes de computadores e por todos os ensinamentos que recebi nesses anos (como fazer pesquisa, como escrever, como apresentar, etc.).

Além dos meus orientadores sou grato aos demais professores do GTA. Agradeço ao Otto Duarte, pelas palavras de sabedoria, alegorias, por aceitar fazer parte da banca e pelas críticas valiosas durante a defesa; ao Pedro Velloso, por ter sido mais colega do que professor; e ao Rodrigo Couto, por ter me ajudado muito desde quando entrei no GTA.

Meus agradecimentos também vão para os demais membros e ex-membros do GTA. Em especial para o Pedro Cruz, Fernando Molano, Dianne Medeiros e Leopoldo Mauricio. O Pedro não cansou de dar inúmeras sugestões nos momentos em que eu empacava, e não cansou de receber meus inúmeros pitacos quando ele era o empacado. O Molano é uma das pessoas mais solícitas que já conheci e também me ajudou incontáveis vezes. A Dianne foi minha colega de sala por boa parte do mestrado e me deu várias lições valiosas. Finalmente o Leopoldo, mesmo sendo aluno parcial, me ajudou a conseguir máquinas para simulação em momento de desespero e me forneceu uma visão prática de alguns dos temas desta dissertação. Agradeço também ao Antonio Silvério, Diogo Mattos, Eric Oliveira, Edvar Afonso, Igor Sanz, Lucas Gomes, Mariana Maciel, Martin Andreoni e Thales Almeida.

Fora do GTA, agradeço aos professores Daniel Figueiredo, José Gabriel Gomes

e Valmir Barbosa pelas excelentes aulas, e aos professores Artur Ziviani e Ítalo Cunha por terem aceitado fazer parte da banca e pelas excelentes críticas e sugestões durante a defesa.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001. Além disso este trabalho contou com apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), da Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ), e da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processos #15/24494-8 e #15/24490-2.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

APRIMORANDO MIDDLEBOXES EM SOFTWARE E ESCALONADORES DE TAREFAS DE DATACENTERS

Hugo de Freitas Siqueira Sadok Menna Barreto

Outubro/2018

Orientadores: Miguel Elias Mitre Campista
Luís Henrique Maciel Kosmalski Costa

Programa: Engenharia Elétrica

Nas últimas décadas, sistemas compartilhados contribuíram para a popularidade de muitas tecnologias. Desde Sistemas Operacionais até a Internet, esses sistemas trouxeram economias significativas ao permitir que a infraestrutura subjacente fosse compartilhada. Um desafio comum a esses sistemas é garantir que os recursos sejam divididos de forma justa, sem comprometer a eficiência de utilização. Esta dissertação observa problemas em dois sistemas compartilhados distintos—*middleboxes* em *software* e escalonadores de tarefas de *datacenters*—e propõe maneiras de melhorar tanto a eficiência como a justiça. Primeiro é apresentado o sistema Sprayer, que usa espalhamento para direcionar pacotes entre os núcleos em *middleboxes* em *software*. O Sprayer elimina os problemas de desbalanceamento causados pelas soluções baseadas em fluxos e lida com os novos desafios de manipular estados de fluxo, consequentes do espalhamento de pacotes. É mostrado que o Sprayer melhora a justiça de forma significativa e consegue usar toda a capacidade, mesmo quando há apenas um fluxo no sistema. Depois disso, é apresentado o SDRF, uma política de alocação de tarefas para *datacenters* que considera as alocações passadas e garante justiça ao longo do tempo. Prova-se que o SDRF mantém as propriedades fundamentais do DRF—a política de alocação em que ele se baseia—enquanto beneficia os usuários com menor utilização. Para implementar o SDRF de forma eficiente, também é introduzida a árvore viva, uma estrutura de dados genérica que mantém ordenados elementos cujas prioridades variam com o tempo. Simulações com dados reais indicam que o SDRF reduz o tempo de espera na média. Isso melhora a justiça, ao aumentar o número de tarefas completas dos usuários com menor demanda, tendo um impacto pequeno nos usuários de maior demanda.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

IMPROVING SOFTWARE MIDDLEBOXES AND DATACENTER TASK SCHEDULERS

Hugo de Freitas Siqueira Sadok Menna Barreto

October/2018

Advisors: Miguel Elias Mitre Campista
Luís Henrique Maciel Kosmalski Costa

Department: Electrical Engineering

Over the last decades, shared systems have contributed to the popularity of many technologies. From Operating Systems to the Internet, they have all brought significant cost savings by allowing the underlying infrastructure to be shared. A common challenge in these systems is to ensure that resources are fairly divided without compromising utilization efficiency. In this thesis, we look at problems in two shared systems—software middleboxes and datacenter task schedulers—and propose ways of improving both efficiency and fairness. We begin by presenting Sprayer, a system that uses packet spraying to load balance packets to cores in software middleboxes. Sprayer eliminates the imbalance problems of per-flow solutions and addresses the new challenges of handling shared flow state that come with packet spraying. We show that Sprayer significantly improves fairness and seamlessly uses the entire capacity, even when there is a single flow in the system. After that, we present Stateful Dominant Resource Fairness (SDRF), a task scheduling policy for datacenters that looks at past allocations and enforces fairness in the long run. We prove that SDRF keeps the fundamental properties of DRF—the allocation policy it is built on—while benefiting users with lower usage. To efficiently implement SDRF, we also introduce live tree, a general-purpose data structure that keeps elements with predictable time-varying priorities sorted. Our trace-driven simulations indicate that SDRF reduces users' waiting time on average. This improves fairness, by increasing the number of completed tasks for users with lower demands, with small impact on high-demand users.

Contents

List of Figures

List of Tables

List of Abbreviations

ACL	Access Control List, p. 24
API	Application Programming Interface, p. 9
ASIC	Application-Specific Integrated Circuit, p. 57
BMF	Bottleneck Max Fairness, p. 47
CDF	Cumulative Distribution Function, p. 14
CPU	Central Processing Unit, p. 1
DDIO	Data Direct I/O, p. 9
DMA	Direct Memory Access, p. 7
DPDK	Data Plane Development Kit, p. 8
DPI	Deep Packet Inspection, p. 5, 16
DRF	Dominant Resource Fairness, p. 3, 11, 30
DSA	Domain-Specific Architecture, p. 57
DSO	Distributed Shared Object, p. 25
ECMP	Equal Cost Multi-Path, p. 3
FIN	TCP flag to indicate the last packet from a sender, p. 17
GPU	Graphics Processing Unit, p. 57
GTA	Teleinformatics and Automation Group (*Grupo de Teleinformática e Automação*), p. v
I/O	Input/Output, p. 3
IP	Internet Protocol, p. 5
IPv4	Internet Protocol, version 4, p. 5

IPv6	Internet Protocol, version 6, p. 5
MMF	Max-Min Fairness, p. 10, 30, 34
NAT	Network Address Translator, p. 2, 5
NFV	Network Function Virtualization, p. 6
NF	Network Function, p. 2, 6
NIC	Network Interface Controller, p. 7
OS	Operating System, p. 7
PO	Pareto Optimality, p. 31
QUIC	Quick UDP Internet Connection, p. 25
RAM	Random-access Memory, p. 21
RFC	Request for Comments, p. 5
RSS	Receive-Side Scaling, p. 2, 9
RST	TCP flag to reset the connection, p. 17
RTT	Round-Trip Time, p. 14
SDRF	Stateful Dominant Resource Fairness, p. 27, 33
SI	Sharing Incentives, p. 31
SMMF	Stateful Max-Min Fairness, p. 34
SP	Strategyproofness, p. 31
SQL	Structured Query Language, p. 10
SYN	TCP flag to synchronize sequence numbers, p. 17
TCP	Transmission Control Protocol, p. 3
TPU	Tensor Processing Unit, p. 57
UDP	User Datagram Protocol, p. 25
UIO	User-space I/O, p. 8
VoIP	Voice over IP, p. 25
YARN	Yet Another Resource Negotiator, p. 10

Chapter 1

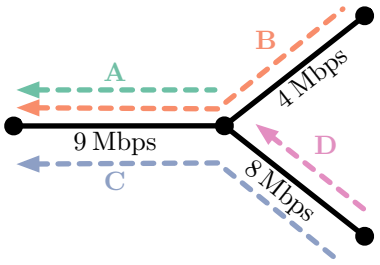
Introduction

Over the last decades, shared systems brought significant cost savings that have contributed to the popularity of many technologies. Packet switching networks allow hosts around the globe to communicate with one another using the same links; operating systems allow multiple processes to use the same CPU; and datacenter schedulers allow tasks from multiple users to run in the same pool of servers. Resource sharing, however, imposes some tradeoffs, such as increasing the control overhead and making achieving consistent performance harder.

Instead of trying to provide performance *guarantees*, most shared systems try to provide performance *isolation* [1–4]. Performance isolation ensures that if a user tries to use too much resources, this has minimal impact on the other users sharing the same system. To provide performance isolation, many shared systems have fairness as their primary objective [3–6]. Fairness can be quantified in a variety of ways, such as Jain's fairness index [7] or Max-Min fairness [1]. The most suitable metric to use depends on the system. A major challenge is that the fairest allocation is often not the most efficient one [8].

To illustrate the fairness-efficiency tradeoff, Figure 1.1 shows an example of multiple flows sharing a network. The example shows different bandwidth allocations obtained when we consider different performance objectives. The first allocation values efficiency and therefore ensures that all the links are fully utilized; however, to do so, it gives flow A a low bandwidth. The second allocation considers Jain's fairness and as such ensures that every flow receives the same bandwidth. Finally, the third allocation considers max-min fairness and is arguably better than the second, since flow D now receives more bandwidth without harming the other flows.

The fairness-efficiency tradeoff presents itself in a variety of ways and in different levels of system design [8–10]. In this thesis we look at how to improve both efficiency and fairness in two distinct systems: software middleboxes and datacenters. In the following sections we present the problems we will investigate in these two systems.



Bandwidth Allocation

Objective	A	B	C	D
Efficiency	1	4	4	4
Jain's Fairness	3	3	3	3
Max-Min Fairness	3	3	3	5

Figure 1.1: Example of bandwidth allocation with different performance objectives for four flows (A, B, C, and D) sharing a network with three links (with 9 Mbps, 8 Mbps, and 4 Mbps). In this example, every flow requires the same bandwidth of 10 Mbps—which is more than what the network is able to provide.

1.1 Efficient Use of Multiple Cores in Software Middleboxes

Today middleboxes are a primary component of both enterprise and Internet provider networks [11, 12]. Middleboxes[1] allow network operators to deploy a wide range of network functions (NFs), such as Network Address Translators (NATs), firewalls, and load balancers. Yet, the cost and lack of flexibility of purpose-built hardware middleboxes are pushing operators to software running on commodity servers [13]. Moving to software, however, does not come for free. Software middleboxes have significant overhead and often need to use multiple CPU cores [14–20]—or even multiple hosts [17, 21–25]—to achieve line rates. Moreover, the rapid increase of network link capacities only exacerbates this need.

When using multiple cores, middleboxes must determine which core to direct packets to. Today, this is often done using Receive-Side Scaling (RSS). RSS is a feature of multi-queue network cards that directs packets to different cores using a hash of the five-tuple (protocol, source and destination IP, source and destination port). Doing so, all packets from the same flow end up in the same core. The reasons for coupling packets from the same flow are twofold. First, processing same-flow packets sequentially avoids packet reordering. Second, having same-flow packets processed in the same core simplifies flow state handling. RSS, however, has significant shortcomings. It is inefficient, since it cannot use all the available cores when the number of concurrent flows is small—which happens frequently in real workloads (§3.1). Moreover, since RSS directs flows to cores using a hash of the five-tuple, hash collisions cause asymmetry in flow distribution.[2] This results in

[1] Middleboxes are devices placed inside the network to perform different functionality than routers and switches. Chapter 2 explains middleboxes in greater depth (§2.1).

[2] Even when the number of cores is comparable to the number of flows, hash collisions happen with high probability due to the birthday problem [26].

unfairness even with a larger number of flows (§3.4).

Interestingly enough, the same problem also appears in a different context. Datacenter networks use per-flow Equal Cost Multi-Path (ECMP) to direct packets to different paths. Similarly to RSS, ECMP directs all packets from the same flow to the same path and, as such, has similar shortcomings [27, 28]. The problems with ECMP have led many [10, 29–32] to consider load-balancing packets to paths ignoring their flows. This approach, known as packet spraying, introduces reordering but, because datacenter networks have paths with low and very similar latencies [33], the amount of reordering is not enough to significantly harm TCP [10]. In face of these similarities, in Chapter 3 we will look for an answer to the following question: *can software middleboxes also improve efficiency and fairness by load balancing packets at a finer granularity?*

1.2 Improving Datacenter Scheduling by Considering Long-Term Fairness

Datacenters are shared by users with different resource constraints [6, 34, 35]. The amount of resources given to each user directly impacts the system performance from both fairness and efficiency standpoints [8]. In single-resource systems, max-min fairness is the most widely used and studied allocation policy [2, 36]. The main idea is to maximize the minimum allocation a user receives. It was originally proposed to ensure a fair share of link capacity for every flow in a network [1]. Since then, max-min has been applied to a variety of individual resource types, including CPU, memory and I/O [2]. Nevertheless, datacenters need to allocate *multiple* resource types at the same time (such as CPU and memory) and max-min is unable to ensure fairness [2, 37].

In a datacenter environment, users often have heterogeneous demands and dynamic workloads [2, 35]. Different mechanisms have been proposed to address the multi-resource allocation problem [2, 37, 38], most notably, Dominant Resource Fairness (DRF) [2]. DRF generalizes max-min to the multi-resource setting, by giving users an equal share of their mostly demanded resource—their *dominant resource*. Using this approach, DRF achieves several desirable properties. Despite the extensive literature on fair allocation, most allocation policies focus only on instantaneous or short term fairness, ensuring that users receive an equal share of the resources regardless of their past behaviors. DRF is no exception, it guarantees fairness only when users' demands remain constant. In practice, however, users' workloads are quite dynamic [35, 39] and ignoring this fact leads to unfairness in the long run [5]. In Chapter 4 we will look for an answer to the following question: *can we improve*

long-term fairness—ensuring that users that use the system sporadically get a greater share of resources—by looking at past allocations?

1.3 Outline

The remainder of this thesis is organized as follows. In Chapter 2 we provide background on software middleboxes and task scheduling in datacenters. In Chapter 3 we present Sprayer, a framework for developing network functions using packet spraying. Then, in Chapter 4 we present SDRF, an extension of DRF that accounts for the past behavior of users and improves fairness in the long run. Finally, in Chapter 5 we conclude the thesis and present conjectures for future work.

The content of this thesis is adapted from our previously published work. The material in Chapter 3 is adapted from [40, 41] and the material in Chapter 4 is adapted from [42].

Chapter 2

Background

This chapter provides background on the topics that will be covered in the chapters that come after. We start with an overview of middleboxes and the technical challenges of moving from specialized hardware to software (§2.1). Then, we delve into the basics of task scheduling in datacenters (§2.2).

2.1 Middleboxes and the Move to Software

In the early days of the Internet, network elements operated in a stateless manner and their functions were limited to simple IP forwarding [43]. This was compatible with one of the fundamental goals of achieving continued connectivity even under the loss of network elements. Internet's popularity boom, however, brought new requirements to the table. For example, the need for improved security led many network operators to deploy firewalls and Deep Packet Inspection (DPI), allowing them to have a finer control over what packets are allowed in their networks, as well as to mitigate possible attacks. These more elaborated network elements are what we call *middleboxes*.

Middleboxes are defined in RFC 3234 [44] as "any intermediary device performing functions other than the normal, standard function of an IP router on the datagram path between a source host and a destination host." Besides improving security, middleboxes can be used to improve performance (*e.g.*, redundancy elimination), provide accountability and monitoring (*e.g.*, traffic monitor), make different protocols compatible (*e.g.*, IPv4 to IPv6 protocol translator), and work around existing limitations (*e.g.*, Network Address Translator – NAT, that allows the Internet to keep scaling in face of the IPv4 address exhaustion).

Although one may argue that middleboxes are fundamentally harmful, breaking the end-to-end principle [45], and that their layer violations make innovation on the Internet harder [46], their popularity is undeniable. In fact, middleboxes are so common today that in some enterprise networks the number of middleboxes is close

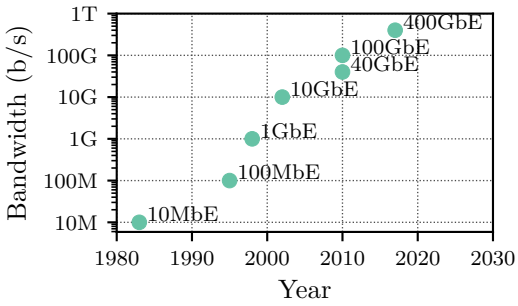


Figure 2.1: Evolution of Ethernet standards.

to the number of routers and switches [11, 12].

2.1.1 The Move to Software

Until recent years, middleboxes were primarily deployed using purpose-built hardware. This, however, has several shortcomings [47]:

- **Rigidity:** Since functionality is implemented directly on hardware, change is very hard—often impossible.
- **Hard to manage:** Middleboxes from multiple vendors have their own management interfaces that do not work together.
- **Slow development:** Hardware is slower and harder to develop than software.
- **Cost:** Some middleboxes are very expensive. Moreover, underutilized boxes offer no opportunity for consolidation.

This started to change in 2012, when major carriers established a cooperation to build what they called Network Function Virtualization (NFV) [13]. NFV aims to solve the above problems by moving middlebox functionality, called Network Functions (NFs), from dedicated boxes to software running on commodity servers. The move to software, however, is not a panacea. Purpose-built hardware generally offers line-rate performance that is challenging to achieve in software. In fact, with the fast adoption of higher-speed Ethernet standards, achieving line rates is only getting harder (see Figure 2.1).

2.1.2 Packet Processing on x86

A straightforward way of implementing NFs using software is to run an application on top of an operating system and leverage its network stack to receive and transmit packets. We will now see that, although this approach works, it is not a good idea from a performance standpoint.

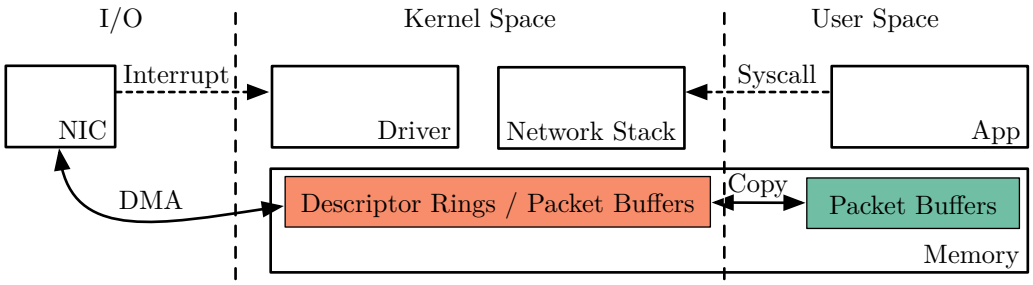


Figure 2.2: Packet processing using Linux network stack.

An application that uses the Linux network stack runs in user space and interacts with the stack in kernel space using system calls (*syscalls*). Figure 2.2 shows an overview of packet processing using the Linux network stack, dashed arrows represent control signals, while solid arrows represent data transfer. When the Network Interface Controller (NIC) receives a packet, it writes it to a buffer in the memory[1] and issues an interrupt. The interrupt triggers the NIC driver's interrupt handler that reads the packet from the buffer and passes it to the network stack. Finally, the network stack parses the packet and copies it to the application's packet buffer in user space. A reverse process happens when the application wants to transmit a packet. The application writes the packet to memory in user space and invokes a syscall. The network stack then copies the packet to a buffer in kernel space and passes it to the driver which informs the NIC that the packet is ready to be transmitted. At last, the NIC reads the packet from memory and sends it to the line, completing the transmission.

The processes above impose significant overheads, most of them due to the separation between user space and kernel space. The first issue is the need to copy packets from one space to another, wasting a considerable number of CPU cycles. Another problem is the need to use syscalls and interrupts to transmit and receive packets. Syscalls and interrupts cause a transition from user space to kernel space, which requires saving the value of some registers to memory. The reason Linux works like this is not because it is unconcerned with performance. Operating systems are designed to make sharing hardware resources possible, the same applies to sharing a NIC. The most common use case for an OS is not packet processing, usually different applications are running at the same time and need to receive and transmit packets. This design makes sure that no application monopolizes the NIC, but the need for better performance in packet processing applications justifies a more restrictive design.

Since most of the overhead imposed by the Linux network stack is due to the

[1]The ability of I/O devices to write and read directly from memory is what we call Direct Memory Access (DMA).

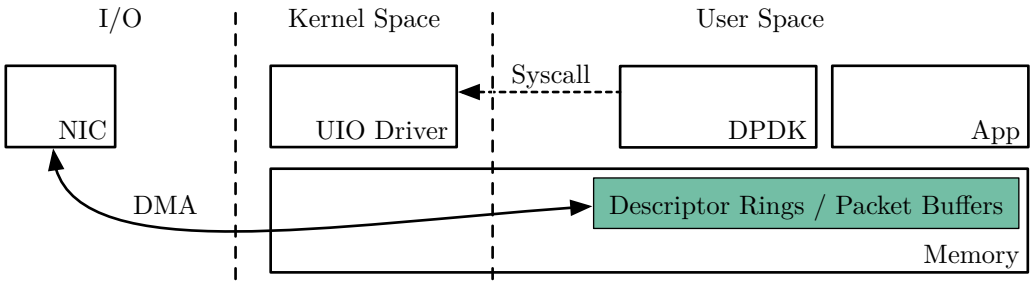


Figure 2.3: Packet processing using DPDK.

separation between user space and kernel space, a natural solution is to do packet processing entirely in the kernel, or entirely in user space. User-space-only packet processing, however, has advantages over kernel-space-only. First, kernel code must be low profile, running fast and yielding the CPU to user-space processes. This is certainly not the case with high-speed packet processing, that often needs multiple dedicated CPU cores (§2.1.3). Second, kernel programming is less flexible, kernel code only has access to a limited set of libraries and is harder to debug.

There are several frameworks for high-performance packet processing in user space, some examples include DPDK [48], netmap [49], and PF_RING ZC [50]. In Chapter 3 we use DPDK for two main reasons: it has better performance [51] and it offers several libraries that aid the development of packet processing applications, *e.g.*, lockless rings and flow classifiers. Figure 2.3 shows an overview of packet processing using DPDK. DPDK bypasses the kernel and communicates directly with the NIC from user space. To do this, it replaces the NIC driver with the UIO (User-space I/O) driver, a minimal driver provided by the Linux kernel to allow the development of drivers in user space. DPDK uses the UIO driver to set up the NIC and map its memory to user space. After the initialization is complete, the NIC reads and writes packets directly to user-space memory and DPDK can configure NIC registers without kernel intervention. There is a problem though, since DPDK now talks directly to the NIC it cannot use the existing kernel drivers, drivers must be implemented inside DPDK. This restricts the set of NICs that can work with it; only NICs that had their drivers ported to DPDK can be used.

Kernel bypass also allows DPDK to avoid the interrupt overhead. Instead of waiting for an interrupt, applications that use DPDK continuously check the memory for new packets. This technique, known as *polling*, wastes CPU cycles when the traffic is low but achieves better performance under heavy loads. Another optimization employed in DPDK codebase is to process batches of packets, instead of individual, whenever possible. These and other optimizations restrain DPDK from being a drop-in replacement for Linux packet socket—which is the way applications that rely on Linux network stack are able to send and receive raw packets [52]. As a

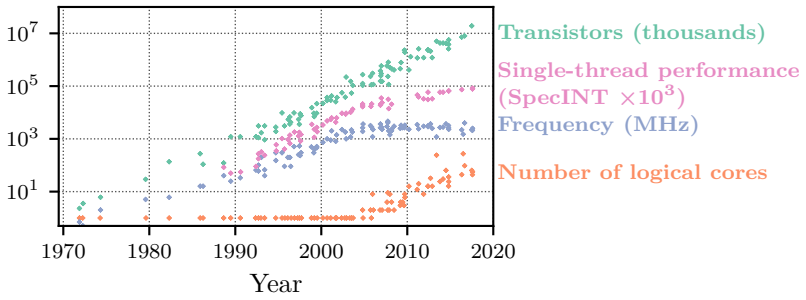


Figure 2.4: Microprocessor trend (adapted from Rupp [53]).

consequence, existing applications that use Linux packet sockets must be rewritten to use the DPDK API.

2.1.3 Using Multiple CPU Cores

Even with all DPDK optimizations, a single CPU core is often not enough to process packets at line rate. Moreover, with faster Ethernet standards (Figure 2.1) and newer CPUs favoring core count over single-thread performance (Figure 2.4), multi-core packet processing is likely to remain the norm in the next years. We will now look at how to extend the design from §2.1.2 to accommodate multiple cores.

As we have seen, the NIC reads and writes packets directly to a packet buffer in the memory.[2] Now that we are using multiple cores, we may think of sharing this packet buffer among all of them. Doing so, however, requires costly synchronization mechanisms. To avoid this problem, we turn to a different solution. Modern NICs have multiple queues that allow them to direct packets to different buffers in the memory. Instead of using a single packet buffer, we associate a different packet buffer to each core. This allows cores to receive and transmit packets independently from one another. Having a separate memory region for each core is also desirable to avoid cache invalidations [54].

Associating packet buffers to cores has a subtle consequence, however. Once the NIC chooses the destination buffer for a packet that arrives, it also chooses the core that is going to process the packet. The NIC commonly makes this choice using Receive-Side Scaling (RSS). RSS was conceived so that packets from the same flow always go to the same buffer. To do this, it decides the destination buffer using a hash of the packet's five-tuple.[3] A problem with hashing flows to cores is that hash collisions occur, and cause significant imbalance in flow distribution, leading to unfairness and inefficiency. This is the problem we will explore in Chapter 3.

[2]In fact, DMA is not the whole story. Modern CPUs use a technology called Data Direct I/O (DDIO) that allow devices to read and write directly to the CPU cache.

[3]The five-tuple consists of five fields from the packet header: protocol, source IP, destination IP, source port, and destination port. A common assumption is to say that packets with the same five-tuple are part of the same flow.

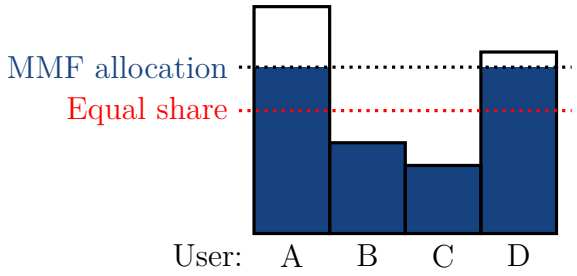


Figure 2.5: Resource allocation among four users using Max-Min Fairness.

2.2 Datacenter Task Scheduling

Clusters of commodity servers have become commonplace. They are responsible for many web services as well as a growing number of data-processing and scientific applications. Yet, managing these clusters is no easy task [55]; cluster managers must ensure good availability in the presence of a high number of failures [56]. To make matters more complicated, clusters are often shared among many users with different requirements and workload types [6, 35]. Examples of cluster managers include open source projects such as Mesos [6] and YARN [34], as well as private solutions, such as Google's Borg [57].

To use a cluster, users submit jobs composed of one or more tasks. Then, the cluster manager is responsible for scheduling these tasks. Workloads differ substantially among users, for example, users that run simulations and machine learning trainings can use the cluster for hours or even days, while some that make interactive SQL queries only need it for a few minutes [6, 35]. In a broad sense, when multiple users share a system, there must be a scheduler that determines the amount of resources each user gets. The requisites for this decision vary among different scenarios. For example, in a public cloud, users pay for the resources they use and fairness is not a concern. In contrast, in a cluster within an institution (research center, lab, or company), usually users do not need to pay for the resources they use. This changes incentives considerably, users want to finish their jobs as fast as possible and, when they need, will try to use the maximum amount of resources [2].

2.2.1 Resource Allocation

Task scheduling involves two different decisions: *which* task to run and *where* to run it. In this thesis we focus on the "which" decision. The scheduler must be able to fairly allocate tasks from different users. One of the most common allocation policies is max-min fairness (MMF). MMF works by giving an equal share of resources for every user; unless a user does not need her entire share, in such case the surplus is divided among the other users. Figure 2.5 shows an example. Users B and C need

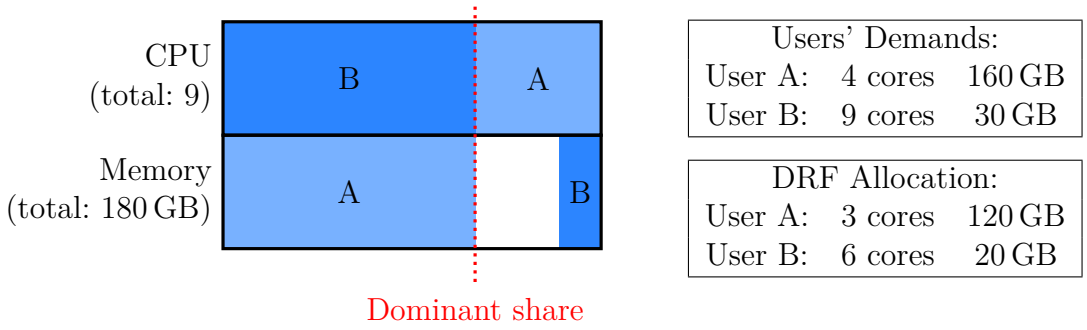


Figure 2.6: DRF allocation for two users with different dominant resources. The share of memory for user A is the same as the share of CPU for user B.

less than their fair share and their surplus is redistributed among users A and D, that need more. This scheme ensures highly-desirable properties:

- **Sharing Incentives:** Being part of the system is at least as good as being part of a hypothetical system with the same amount of resources but where every user has a proportional and exclusive share.
- **Strategyproofness:** Users cannot improve their allocations by misreporting their demands to the scheduler.
- **Pareto Optimality:** Resources are allocated in such a way that it is impossible to increase the allocation of a user without decreasing the allocation of another.

MMF works well when demands are static, however in practice, demands are quite dynamic and users with long running jobs coexist with users that have sporadic short jobs [6, 35]. As we will see in Chapter 4, MMF fails to ensure fairness in the long run, resulting in users with long running jobs benefiting more from the system than those that use the system sporadically. Moreover, as we will discuss next, MMF can only be used when the system has a single resource type.

2.2.2 Multiple Resource Types

So far, in our discussion on resource allocation, we considered that a single resource type is being shared. Nevertheless, when scheduling tasks, usually multiple resources are shared (*e.g.*, CPU and memory), which makes MMF unsuitable.

Dominant Resource Fairness (DRF) [2] is a notable policy[4] that extends MMF to multiple resource types. To do this, it uses the concept of *dominant resource*. Dominant resource is the resource a user needs the most relative to the total in the system. For example, if a system has a total of 10 CPU cores and 100 GB

[4]Both Mesos and YARN implement a DRF scheduler.

of memory, the dominant resource for a user that needs 5 CPU cores (50%) and 20 GB of memory (20%) is CPU. When allocating resources, DRF tries to equalize users' share of dominant resource (*dominant shares*). Figure 2.6 shows an example of DRF allocation when two users share a system with two types of resources. User A's dominant resource is memory, while user B's is CPU. DRF gives each user the same dominant share. More broadly, the DRF allocation can be obtained by applying MMF to users' dominant shares. This means that if a user needs less than her dominant share, DRF will reallocate the surplus among the other users. An important aspect of DRF is that it inherits the MMF properties listed in §2.2.1. Moreover, when there is a single resource type, DRF reduces to MMF. By being an extension to MMF, DRF also has problems to ensure fairness in the long run. In Chapter 4 we introduce an allocation policy that addresses these problems while ensuring the same properties.

Chapter 3

Sprayer

In this chapter we present Sprayer, a framework for developing network functions using packet spraying. Packet spraying solves the imbalance problems caused by RSS, but makes flow state handling more challenging. Sprayer uses features of commodity NICs to spray packets to cores without software intervention. Moreover, it equips NFs with abstractions for handling flow states. Sprayer's flow state abstractions build on the observation that most NFs only update flow state when TCP connections start or finish (§3.2.2). Therefore, by directing packets at the beginning or end of the same TCP connection (*connection packets*) to the same core, we ensure that only this core will need to modify the state for this connection. This avoids the introduction of synchronization primitives that would impact performance.

We conduct experiments to understand how effective Sprayer is in comparison to RSS. Similarly to the datacenter observations, we find that the low difference in delay between packets processed in different cores is not enough to significantly impair TCP performance. Moreover, we observe that the overall TCP throughput remains consistent for both low and high number of concurrent flows. Therefore, for the typical number of concurrent flows found in real workloads, Sprayer greatly improves TCP throughput, compared to RSS. Further, we show that Sprayer also improves fairness, even with a higher number of flows.

This chapter is organized as follows. In §3.1 we use real packet traces to motivate the need for packet spraying in middleboxes. Then, we detail Sprayer's design in §3.2, and implementation in §3.3. We conduct experiments to evaluate Sprayer in §3.4 and discuss its limitations in §3.5. Finally, we survey related work in §3.6 and conclude the chapter in §3.7.

3.1 Motivation

To motivate the need for packet spraying in middleboxes, we begin with a quick analysis of real packet traces. We want to understand how diverse is the traffic at

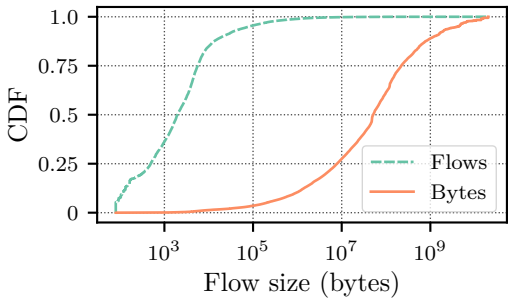


Figure 3.1: Distribution of number of flows with a given size and distribution of bytes across different flow sizes.

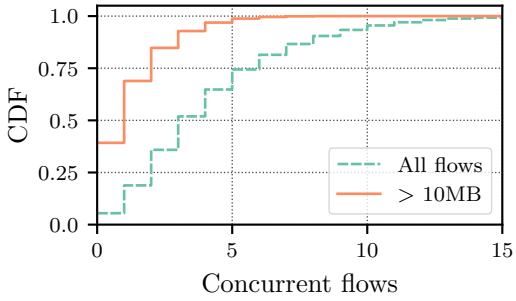


Figure 3.2: Number of concurrent flows in every 150 µs window, considering all flows or only large flows.

the small time frame that packets stay inside a middlebox.

We use a 48 h trace of a highly-utilized 1 Gbps backbone link [58] captured in May 2018. The trace does not contain packet payloads, we determine packet sizes using the "Total Length" field of the IP header. Figure 3.1 shows the CDF of TCP flow sizes as well as the distribution of bytes across these flows. There are few large flows, but they are responsible for the majority of the traffic. Flows with more than 10 MB account for more than 75% of the traffic. This confirms the long observed "elephants and mice" phenomenon of Internet traffic [59].

The effectiveness of RSS on middleboxes depends on the number of concurrent flows. If this number is large enough, RSS uses all cores with high probability. Although the number of ongoing TCP connections can be very large,[1] if we consider only the number of flows active in the small amount of time it takes for a packet to be processed by a middlebox, this assumption no longer holds.

To measure concurrent flows, we use a 150 µs window. This window is 10 times the largest 99$^{\text{th}}$ percentile RTT we found in our experiments (§3.4).[2] This RTT is also comparable to the one measured by previous work [19, 47, 60]. Since the actual time a packet takes to be processed by the middlebox is certainly less than the RTT,

[1]The number of ongoing TCP connections can exceed 1 million in this trace.

[2]This RTT is measured using a server directly connected to the middlebox. We explain this experiment in §3.4.

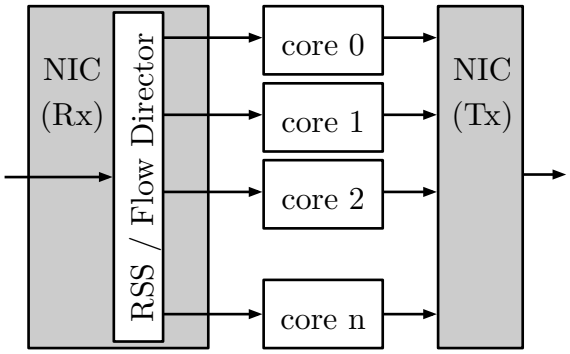


Figure 3.3: Hardware packet classification. The NIC is responsible for directing packets to cores.

the number of concurrent flows we report is a strict upper bound.

Figure 3.2 presents the CDF of the number of concurrent TCP flows. The median number of concurrent flows is only 4 and the 99$^{\text{th}}$ percentile is 14. The level of concurrency among large flows is even smaller. If we only consider flows with more than 10 MB, the median number of concurrent flows is 1 and the 99$^{\text{th}}$ percentile is 6. Yet, as we have seen, these flows account for the majority of the traffic, which indicates a poor degree of statistical multiplexing.

Since these results are for a backbone link, we expect them to include more concurrent flows than the traffic of an enterprise network. Indeed, we repeated the same analysis on traffic at our lab's Internet gateway and on the M57 traces [61] (used by some previous work on middleboxes [18, 23]) and found even fewer concurrent flows.

3.2 Design

We now turn to the design of Sprayer. First we describe the challenges of processing sprayed packets. Then we present an architecture that deals with these challenges. Finally, we delve into a simple programming model used by NFs implemented on top of Sprayer.

There are two main challenges in the design of Sprayer: spraying packets to different cores and handling flow states.

3.2.1 How to spray packets?

When processing packets in a multi-core system, one has to choose between software and hardware packet classification. As depicted in Figure 3.3, the hardware technique consists of using multi-queue NICs, which are common today, to classify and direct packets to each core. The software alternative is to direct all packets to a sin-

Table 3.1: Example of state scope and access pattern of some popular stateful NFs. Most NFs only update flow states when connections start or finish.

NF	**State**	**Scope**	**Access Pattern**	
			packet	**flow**
NAT, IPv4 to IPv6	Flow map	Per-flow	R	RW
	Pool of IPs/ports	Global	-	RW
Firewall	Connection context	Per-flow	R	RW
Load Balancer	Flow-server map	Per-flow	R	RW
	Pool of servers	Global	-	RW
	Statistics	Global	RW	-
Traffic Monitor	Connection context	Per-flow	-	RW
	Statistics	Global	RW	-
Redundancy Elimination	Packet cache	Global	RW	-
DPI	Automata	Per-flow	RW	-

gle core and let software choose the destination cores. Using hardware classification offers better performance and is usually the preferred method [11, 18]. Since current NICs do not offer support for spraying packets to cores, one might be tempted to turn to software-based classification. Fortunately, we discovered a way of spraying packets using Flow Director, a functionality found in many commodity NICs. We delay the implementation details to §3.3. For now, it is sufficient to know that the NIC randomly delivers TCP packets to cores.

3.2.2 How to handle flow state?

The traditional approach of sending all the packets from the same flow to the same core has the benefit that flow states are partitionable and each core only has to keep state for its flows. Partitionable state is often desirable as it avoids the penalty of enforcing cache coherence, as well as the use of synchronization primitives (*e.g.*, locks). When we blindly spray packets from the same flow across all cores, we lose this property. What we observe, however, is that we get similar benefits if we only provide *writing* partition. As long as we guarantee that the state of a given flow is only modified by a single core, we avoid the use of locks and significantly reduce cache invalidations.

In order to provide writing partition, we depart from the observation that most NFs only change flow state when TCP connections start or finish. Table 3.1 shows the scope (per-flow or global state) and access pattern (read or write at every packet or flow) for some popular stateful NFs. Deep Packet Inspection (DPI) is the only NF in the list that needs to update flow state for every packet. Of course, some NFs also need to update *global* state for every packet. Although this issue also has the

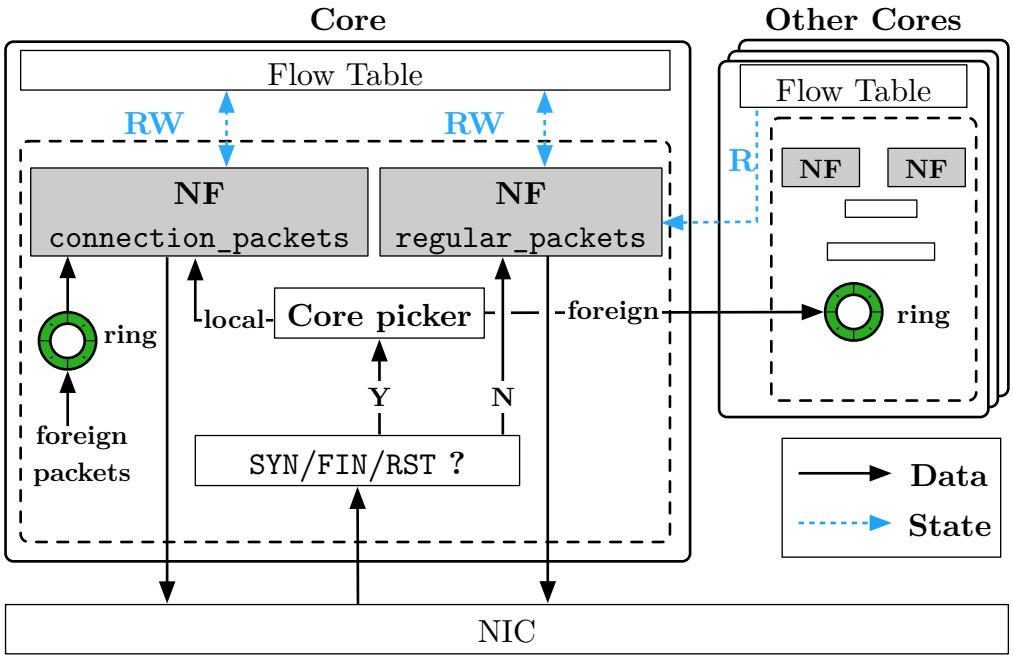


Figure 3.4: Overview of Sprayer from the perspective of a single core. Regular packets are processed locally, while connection packets may be transferred to other cores.

potential to affect performance, it is not specific to Sprayer, traditional approaches must also deal with shared global state. Moreover—at least in the case of statistics—looser consistency is often tolerable, which helps to reduce the problem [25].

We make a distinction between *connection packets* and *regular packets*. Connection packets are those that have potential to modify TCP state (packets flagged with `SYN`, `FIN`, or `RST`), while regular packets are all the others. Moreover, we say that every flow has a *designated core*. We determine the designated core for a given flow calculating a hash of its five-tuple. By default, we use a hash function that maps upstream and downstream flows from the same TCP connection to the same designated core. Sprayer enforces writing partition by keeping flow states in their designated cores while making sure that all connection packets from a given flow are processed in their designated core.

3.2.3 Architecture

Figure 3.4 shows an overview of Sprayer's architecture. The key idea is to separate the NF code that handles connection packets from the code that handles regular ones. All cores run identical threads and have their own flow tables. Moreover, cores can only write to their local flow tables, but can read from any. This ensures writing partition.

After the NIC delivers a packet, Sprayer checks whether it is a connection packet.

Table 3.2: Flow state API.

Function	Description
`insert_local_flow(flow_id)`	Insert flow entry in local table
`remove_local_flow(flow_id)`	Remove flow entry from local table
`get_local_flow(flow_id)`	Retrieve modifiable flow entry from local table
`get_flow(flow_id)`	Retrieve unmodifiable flow entry from its designated core

It then processes regular packets in the core they arrive but redirects connection packets to ring buffers in their designated cores—unless the designated core is the same as the current one (core picker in Figure 3.4). Note that Sprayer does not transfer the entire packet to other cores, it transfers packet descriptors. Packet descriptors contain information about a particular packet, including its memory address. Also note that if NICs were able to deliver connection packets to cores based on their five-tuples, while spraying the others, Sprayer would not need to transfer those packets.[3]

For performance reasons, we use batches of packets whenever possible. For example, if we need to transfer more than one packet to the same core, we send them all together in a batch. Moreover, segregating the code that handles connection packets from the code that handles regular packets allows us to deliver batches of pre-classified packets to these functions. In the case of the function that processes connection packets, packets from both local and foreign cores can be placed in the same batch. This segregation also makes sense from an NF programmer's standpoint, as we will see next.

3.2.4 Programming Model

An NF built using Sprayer must implement two packet-handler functions. The `connection_packets` function receives connection packets and therefore contains logic to deal with opening or closing connections. As it is guaranteed to receive all connection packets for a given flow, it can store state for this flow in its local flow table. Later, since the designated core is deterministic, a `regular_packets` function from any core that needs this state knows where to look.

Sprayer abstracts flow state accesses with its flow state API (Table 3.2). There are functions to remove or insert state in the local flow table as well as to retrieve local or global flow states. Only local states are modifiable. When the NF calls `get_flow` with a specific flow id, Sprayer determines its designated core and retrieves

[3]Although this is not possible with commodity hardware, it is an opportunity for future work (see §3.5).

the flow state from its flow table. Note that the *constness* of the flow entry returned by the `get_flow` function is only lightly enforced, we use a C pointer to a `const` variable, that means that a programmer may remove the *constness* and modify the variable. Although removing this *constness* is possible, it may cause undefined behavior, and on some situations triggers compiler warnings. Besides the functions in Table 3.2, Sprayer has an optimized version of `get_flow` for looking up multiple flow states at a time.

Of course, there is much more complexity in programming an NF than flow state access. Our focus here is not in providing a comprehensive set of tools for NF programming—others have done it already [47, 62, 63]—instead we argue that Sprayer's flow state abstractions are simple to use and can be incorporated to other solutions.

We use a simple implementation of a NAT to demonstrate how to use Sprayer's flow state abstractions (Figure 3.5). For brevity, we only consider TCP packets, and omit variable declarations and flow removal logic. Moreover, a real implementation will use batches of packets instead of separately handling each. The `connection_packets` function, upon receiving the first `SYN` packet from a TCP connection, selects a port from a global pool (line 10) and uses `insert_local_flow` to save this translation in the local flow table (lines 18–19). Since the designated core is the same for both sides of the same TCP connection, the NAT can also store the translation for the other side (lines 25–26). NAT then treats all the packets that come after (including `SYN-ACK`) as regular packets. The `regular_packets` function only has to retrieve the translation using `get_flow` (line 31) and use it to update the packet header (line 39). Sprayer API also helps NFs that need to record statistics but tolerate looser consistency. These NFs can keep statistics for all flows in every core and periodically aggregate them in their designated cores.

In addition to packet handlers, Sprayer also allows NFs to implement an initialization function. Besides initialization work (*e.g.*, memory allocation), NFs can use this function to set parameters that Sprayer will use in its own initialization, such as the size of the flow table and its entries. Stateless NFs can also set a flag to disable flow state features, *i.e.*, flow tables and the redirection of connection packets.

3.3 Implementation

We have implemented Sprayer on top of DPDK [48], taking advantage of many state-of-the-art optimizations, such as polling and batching. In order to make the NIC spray packets we also had to modify DPDK's `ixgbe` driver [64]. At first glance, it may seem impossible to spray packets using existing commodity NICs, since they do not offer this functionality [65, 66]. We, however, circumvent this limitation using

```
1 void connection_packets(pkt_t* pkt) {
2     // we only care about the first SYN packet
3     if (!pkt->SYN || pkt->ACK) {
4         regular_packets(pkt);
5         return;
6     }
7     flow_id = get_five_tuple(pkt);
8
9     // select a port from pool
10    translated_flow_id = select_port(flow_id);
11
12    // no port available or invalid source IP
13    if (!translated_flow_id) {
14        drop_packet(pkt);
15        return;
16    }
17
18    flow_entry = insert_local_flow(flow_id);
19    *flow_entry = translated_flow_id;
20
21    update_header(pkt, translated_flow_id);
22
23    // we also include the other side
24    rev_flow_id = reverse(translated_flow_id);
25    flow_entry = insert_local_flow(rev_flow_id);
26    *flow_entry = reverse(flow_id);
27 }
28
29 void regular_packets(pkt_t* pkt) {
30    flow_id = get_five_tuple(pkt);
31    translated_flow_id = get_flow(flow_id);
32
33    // no translation found for this flow id
34    if (!translated_flow_id) {
35        drop_packet(pkt);
36        return;
37    }
38
39    update_header(pkt, translated_flow_id);
40 }
```

Figure 3.5: Sample implementation of a NAT. Sprayer's API functions and packet handlers are underlined.

Flow Director [65], a feature of Intel NICs designed to associate *specific* sets of flows to queues. We use Flow Director in an unconventional manner: instead of matching flows, we configure it such that packets are directed to queues using the checksum field of the TCP header. Since the checksum field looks random, TCP packets are uniformly distributed across queues regardless of their flows. In contrast, non-TCP packets fail to match any rules and fall back to traditional RSS, in which the NIC directs packets to cores using a hash of the five-tuple. All non-TCP packets are processed in the core they arrive, with no need for redirection.

A major problem with Flow Director—and in fact the reason many choose not to use it [15, 60]—is that it has a somewhat limited space for flow rules (8k). We avoid this problem using only a certain number of least significant bits of the checksum field, depending on the number of cores in the system. This allow us to define rules that exhaust all possible matches.

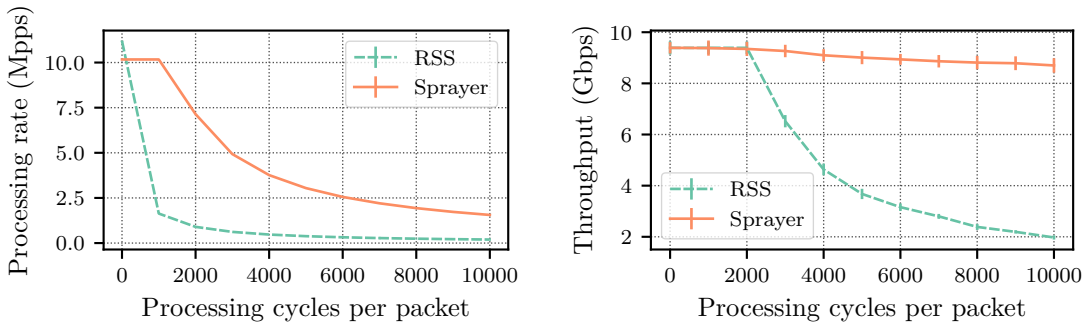
3.4 Evaluation

This section presents an evaluation of Sprayer. We run experiments on a testbed with two servers connected back-to-back. One server functions as a traffic generator and the other as a middlebox. The middlebox server is equipped with two Intel Xeon E5-2650 CPUs, each of which has 8 cores with 2.0 GHz clock, and 256 GB of RAM (equally divided among all memory channels). The traffic-generator server is equipped with a single Intel Core i7-7700 CPU and 32 GB of RAM. Moreover, both servers have an Intel 82599ES 10 GbE NIC [65] and run Linux 4.9.0-5. We configure the RSS hash function to direct upstream and downstream flows from the same connection to the same core [67].

To systematically emulate NFs with different complexities, we implement a simple NF on top of Sprayer. This NF creates a new entry in the flow table at every new connection. Moreover, for every packet it receives, it retrieves the flow state, modifies the header, and busy loops for a given number of cycles. We vary the number of cycles from 0 up to 10,000 (the maximum number of cycles per packet among the NFs surveyed by [20]). The NF uses 8 cores in all experiments.

When measuring processing rate, we use MoonGen [68] to generate 64 B TCP packets with variable payload content, and therefore variable checksum. When measuring TCP throughput, we use Iperf3 [69] to create real TCP connections. Our results use the standard Linux TCP implementation (CUBIC), without any kind of tuning. Unless otherwise noted, error bars represent one standard deviation.

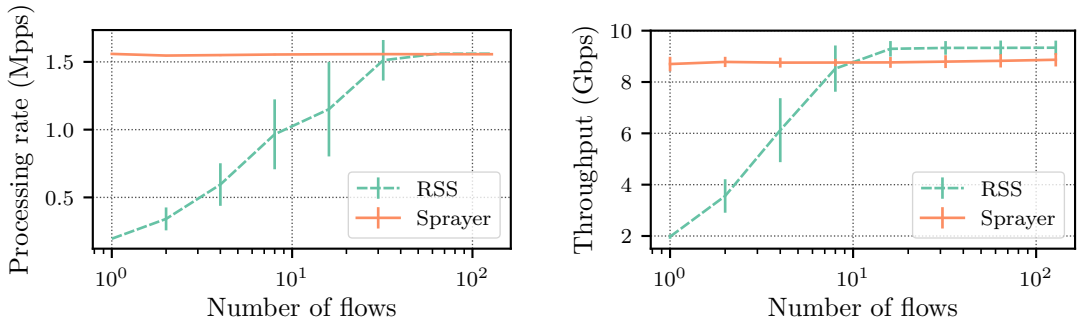
How much can Sprayer improve performance? The maximum improvement caused by Sprayer happens when there is a single flow. Figure 3.6a shows the



(a) Processing rate.

(b) TCP throughput.

Figure 3.6: Effect of increasing the number of processing cycles per packet on processing rate (with 64 B packets) and TCP throughput, while using a single flow.



(a) Processing rate.

(b) TCP throughput.

Figure 3.7: Effect of increasing the number of flows on processing rate (with 64 B packets) and TCP throughput. Processing cycles per packet remain fixed at 10,000.

processing rate as a function of per-packet processing cycles for a single flow. As expected, when we increase the number of cycles spent on each packet, the processing rate decreases. Somewhat unexpectedly though, Sprayer's processing rate is limited to about 10 Mpps. This, however, is not fundamental and is a limitation of the 82599 NIC when using Flow Director. For less trivial NFs, the fact that Sprayer uses all cores allows it to process significantly more packets than RSS. Since Sprayer may reorder packets, improving processing rate does not necessarily improve TCP throughput. Figure 3.6b alleviates this concern by measuring the throughput of a real TCP connection.

How does the number of flows impact Sprayer? The performance of Sprayer is consistent regardless of the number of concurrent flows. We repeat the same experiments fixing the number of processing cycles per packet in 10,000 while increasing the number of flows. Sources and destinations change randomly at every execution. When generating packets from different flows using MoonGen, we use round-robin, so that packets from different flows are perfectly interleaved—this is the best-case scenario for RSS. Figure 3.7 compares the processing rate and TCP

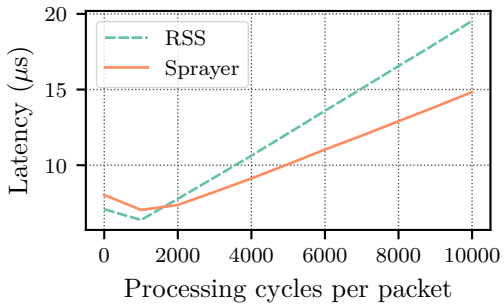


Figure 3.8: 99$^{\text{th}}$ percentile RTT for 64 B packets at 70% load for a single flow.

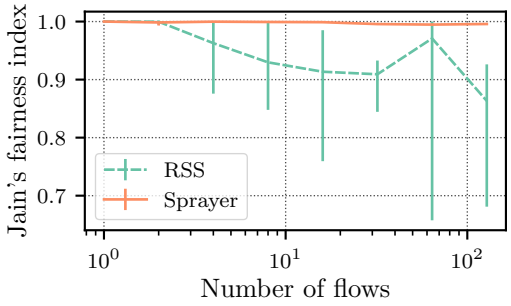


Figure 3.9: Jain's fairness index for an increasing number of flows.

throughput of RSS and Sprayer, for increasing numbers of concurrent flows. We find that RSS shows considerably worse throughput for a small number of flows and a slightly better throughput for a sufficiently large number of flows. Since the processing rate between the two is similar for a large number of flows, we attribute the difference in TCP throughput to packet reordering. Furthermore, if we consider the small number of concurrent flows in a typical workload (Figure 3.2), Sprayer is faster most of the time. Also note, that the measurements for RSS have larger error bars. That is because hash collisions change from one experiment to another, causing better or worse throughput.

Does Sprayer impact latency? Since Sprayer spreads packets from the same flow across all cores, packets from the same flow are processed in parallel. This ends up reducing latency. Figure 3.8 compares the 99$^{\text{th}}$ percentile round trip time when using RSS and Sprayer to process 64 B packets from a single flow at 70% of the minimal processing rate between RSS and Sprayer.

Does Sprayer impact fairness? Sprayer eliminates the fairness problem caused by hash collisions. Since all flows get to share all cores equally, they all receive the same share. Figure 3.9 reports the average Jain's fairness index [7] across all runs. Error bars represent the minimum and maximum observations. While Sprayer consistently achieves fair throughput (Jain's index close to 1.0), RSS's fairness depends on the number of flows each core has to process.

Summary. Our experiments indicate that spraying packets across cores is a valid approach for software middleboxes. It improves fairness and provides consistent performance, regardless of the number of flows. What remains to be answered is how well other TCP implementations interact with the levels of packet reordering imposed by Sprayer. Moreover, although the NF used in our experiments operates similarly to a real NF,[4] we plan to extend our evaluation to real NFs implemented on top of Sprayer.

3.5 Discussion

We now point to Sprayer's limitations and outline questions that should be further investigated.

NF deployability: Sprayer's programming model can be used to implement NFs that do not need to update flow state in the middle of a flow (*e.g.*, NAT, firewall, load balancer, traffic monitor). However, not every NF fits this model. Some DPIs, for example, need to support cross-packet pattern matching. Although they can be made to work with out-of-order packets [70], implementing them on top of Sprayer would require that cores share their state machines. Another example of NFs incompatible with Sprayer are transparent web proxies and caches. The reason being that an HTTP request may be split among different TCP packets and end up going to different cores. Since transparent proxies are incompatible with HTTPS—which now accounts for more than 70% of loaded web pages [71, 72]—we do not see this as a major drawback.

Programmable NICs: We constrained our design to work on commodity hardware. However, the rise of programmable NICs [73–75] creates further opportunities. First, we could program NICs to direct connection packets to designated cores, reducing some of Sprayer's overhead. Also, inspired by previous work on datacenter networks [76–78], we may configure NICs to direct packets to cores using *flowlets*. Flowlets are a middle ground between packets and flows. They are based on the observation that packets from the same flow often arrive in bursts. Datacenters that use flowlets direct these bursts of packets to the same path. This can bring advantages, such as reduced packet reordering.

Scalability with more cores: Although an increase in the number of CPU cores should increase Sprayer's advantage over RSS, it also has the potential to increase packet reordering. Therefore, it may be wise to only spray packets from a particular

[4] Our NF does a flow-state lookup, updates the header, and busy-loops for a certain number of cycles. A firewall, for example, would lookup the flow state and go through an access control list (ACL).

flow to a limited subset of cores [79]. We intend to test this hypothesis in future work using programmable NICs.

Elastic scaling to multiple hosts: In this work we focused on improving utilization of a single host. In some situations, however, NFs need to scale to multiple hosts [17, 23–25]. We can also scale Sprayer to multiple hosts, as long as packets from the same flow are not sprayed across different hosts. Moreover, proposals like S6 [25], that advocates using a Distributed Shared Object (DSO) to share state among hosts, could also be used to scale Sprayer.

Different transport protocols: At our current implementation, Sprayer only sprays TCP packets; other packets continue to be directed to cores using RSS. This avoids the potential problems packet reordering causes to some UDP applications (*e.g.*, VoIP [78]). More elaborated classification could be made to spray only *some* UDP flows. QUIC [46], for example, runs on top of UDP and by design is more resilient to packet reordering than TCP.

3.6 Related Work

As already mentioned, there are multiple works that use packet spraying to improve both efficiency and fairness in datacenter networks [10, 29–32]. Yet, Sprayer is the first to bring this concept to software middleboxes. Although the basic idea is similar, the implications are different. One of the challenges of using packet spraying in datacenters is to ensure that it keeps working in the presence of asymmetries caused by link failures. In middleboxes, this problem does not exist. Instead, flow state sharing is the main concern.

Many previous works have also investigated NF state so as to scale NFs to multiple hosts [17, 22–25]. Despite these solutions being orthogonal to our work, they have identified similar flow-state-access patterns as we did. Moreover, one of these solutions, StatelessNF [23], moves all NF state (per-flow and global) to a remote server, which is an elegant approach to simplifying scalability and failure recovery. Although StatelessNF could potentially replace Sprayer's flow state abstractions, it requires non-commodity technology (InfiniBand). Moreover, accessing remote states increases latency and requires extra CPU cycles [25].

Some attempts have also been made to improve middlebox efficiency when packets need to go through multiple NFs (NF chaining). Solutions such as NFP [19] and ParaBox [80] explore parallelism by processing the same packet in NFs located in different cores at the same time. These solutions, however, are specific to NF chaining and can only work for some configurations. Moreover, they require at least two inter-core transfers for every packet. Also related to NF chaining, NFVnice [16]

tries to improve fairness *among NFs* running on the same core, but makes no effort to improve fairness *among flows*.

Finally, mOS [62] has focused on creating abstractions for stateful flow processing. It keeps track of TCP state machines and let NFs implement handlers, which are triggered in the presence of events (*e.g.*, new TCP connection). This is complementary to Sprayer's flow state abstractions, that facilitate flow state access in the presence of packet spraying.

3.7 Conclusion

In this chapter, we introduced Sprayer. Sprayer allows NFs to load balance packets to multiple CPU cores using packet spraying, instead of flow-based hashing. It also provides abstractions for handling flow state without the need for synchronization primitives. We observed that, when compared to the per-flow alternative, Sprayer significantly improves fairness and consistently uses the entire capacity, even when there is a single flow.

Chapter 4

Stateful Dominant Resource Fairness

In this chapter we introduce Stateful Dominant Resource Fairness (SDRF), an extension of DRF that accounts for the past behavior of users and improves fairness in the long run. The key idea is to make users with lower average usage have priority over users with higher average usage. When scheduling tasks, SDRF ensures that users that only sporadically use the system have their tasks scheduled faster than users with continuous high usage. The intuition for SDRF is that when users use more resources than their rightful share of the system, they commit to use less in the future if another user needs. SDRF tracks users commitments and ensures that whenever system resources are insufficient, commitments are honored.

We conduct a thorough evaluation of SDRF and show that it satisfies the fundamental properties of DRF. SDRF is strategyproof as users cannot improve their allocation by lying to the mechanism. SDRF provides sharing incentives as no user is better off if resources are equally partitioned. Moreover, SDRF is Pareto efficient as no user can have her allocation improved without decreasing another user's allocation. DRF can be efficiently implemented using a priority queue that determines which user has the highest allocation priority. When we consider the past, allocation priorities may change at any instant and the implementation cannot benefit from a priority queue. We mitigate this problem—being able to implement SDRF efficiently—introducing live tree, a data structure that keeps elements with predictable time-varying priorities sorted.

Besides the theoretical evaluation, we analyze SDRF using large-scale simulations based on Google cluster traces containing 30 million tasks over a one-month period, and compare it to regular DRF. Results show that SDRF reduces the average time users wait for their tasks to be scheduled. Moreover, it increases the number of completed tasks for users with lower demands, with negligible impact on high-demand users. We also use Google cluster traces to evaluate the performance

of live tree, concluding that SDRF works well in practice.

This chapter is organized as follows. We introduce the system model in §4.1 and use it to define DRF and its allocation properties in §4.2. We then introduce SDRF and show its properties in §4.3. In §4.4 we focus on the implementation of SDRF using a live tree. We then test SDRF and our implementation under trace-driven simulations in §4.5. Finally, we review related work in §4.6 and conclude the chapter in §4.7.

4.1 System Model

In this section, we model the multi-resource allocation problem in a multi-user system. We first formalize users and resource demands, and then define the general structure of an allocation mechanism. From this structure we formalize users' sequential interactions as a repeated game.

4.1.1 Multi-Resource Setting and Allocation Mechanism

The system consists of a set of users $\mathcal{N} = \{1, \ldots, n\}$ that share a pool of different hardware resources $\mathcal{R} = \{1, \ldots, m\}$. Without loss of generality, we normalize the total amount of every resource in the system to 1, *i.e.*, if a system has a total of 100 CPU cores and 10 TB of memory, 0.1 CPU equals 10 cores while 0.1 memory equals 1 TB. For simplicity, we assume that the set of users and the amount of resources remain fixed. Every user i has a demands vector $\boldsymbol{\theta}_i^{(t)} = \langle \theta_{i1}^{(t)}, \ldots, \theta_{im}^{(t)} \rangle$ representing the user demand for every resource at instant t. We consider positive demands for every resource type,[1] therefore at every instant t, $\theta_{ir}^{(t)} > 0, \forall i \in \mathcal{N}, r \in \mathcal{R}$.

The allocation mechanism should produce as output a resource allocation based on users' *declared demands*. We represent the declared demands vector for a user i at instant t analogously to the demands vector, $\hat{\boldsymbol{\theta}}_i^{(t)} = \langle \hat{\theta}_{i1}^{(t)}, \ldots, \hat{\theta}_{im}^{(t)} \rangle$. When users declare demands truthfully, $\hat{\boldsymbol{\theta}}_i^{(t)} = \boldsymbol{\theta}_i^{(t)}$. We also define the allocation vector for user i at instant t for every resource type as $\boldsymbol{o}_i^{(t)} = \langle o_{i1}^{(t)}, \ldots, o_{im}^{(t)} \rangle$. The allocation returned by the mechanism at instant t is represented by a matrix of all the individual allocation vectors: $\boldsymbol{O}^{(t)} = \langle \boldsymbol{o}_1^{(t)}, \ldots, \boldsymbol{o}_n^{(t)} \rangle$. We impose a feasibility restriction to the allocations so that they may never be greater than the total amount of resources in the system, *i.e.*, at every instant t, $\sum_{i \in \mathcal{N}} o_{ir}^{(t)} \leq 1, \forall r \in \mathcal{R}$.

We represent user's preferences using a utility function. Given an arbitrary

[1] The requirement of non-zero demands is to avoid problems in the model. In practice, users may not need every resource type at every instant. We can still use the same model and say that these users need ϵ resources, where ϵ is an arbitrarily small positive quantity.

allocation $\boldsymbol{o}_i^{(t)}$, for every user i and time t, the utility function is

$$u_i^{(t)}(\boldsymbol{o}_i^{(t)}) = \min\left\{\min_{r\in\mathcal{R}}\{o_{ir}^{(t)}/\theta_{ir}^{(t)}\}, 1\right\}. \tag{4.1}$$

Intuitively, users prefer allocations that maximize their number of tasks, being indifferent between different allocations that result in the same number of tasks (when the utility is 1, the user is able to allocate all the tasks she desires). This assumes tasks are arbitrarily divisible [2, 9, 81]. This assumption does not hold in practice and we evaluate its impact in §4.4.2. Note that we do not rely on the utility function for interpersonal comparison, we only use it to induce ordinal preferences [81, 82]. This means that, even though the utility function can be used to determine which allocation is better for a user, it cannot be used to determine if one user is doing better than another.

4.1.2 Repeated Game

In the previous sub-section we referred to an instant t when defining most notations, however we omitted the influence time has in the allocation and in the user's preferences. In game theory, we typically say that at every instant t there is a *stage game* where users declare their demands $(\hat{\boldsymbol{\theta}}_i^{(t)}, \forall i \in \mathcal{N})$ and the allocation mechanism decides an allocation $(\boldsymbol{o}_i^{(t)}, \forall i \in \mathcal{N})$. The sequence of stage games defines the *repeated game*. To evaluate user's expected long-term utility, we consider that they discount future utilities using a discount factor $\delta_i \in [0, 1)$, *i.e.*, user i's *expected long-term utility* in the repeated game for the instant t is

$$u_i^{[t,\infty)} = \mathbb{E}_{u_i}\left[(1-\delta_i)\sum_{k=t}^{\infty}\delta_i^{k-t}u_i^{(k)}(\boldsymbol{o}_i^{(k)})\right]. \tag{4.2}$$

The normalization factor $(1-\delta_i)$ adjusts the units so that we can compare the stage-game and repeated-game utilities.[2] The discount factor δ_i is often called the "user patience"; the closer it is to 1, the more users care about future outcomes. Conversely, the closer it is to 0, the more users care about recent future and the stage-game outcomes. Table 4.1 has a summary of all the notations used in this chapter.

4.2 DRF and Allocation Properties

In this section, we quickly review the DRF mechanism and the static allocation properties DRF and DRF-based schedulers usually satisfy. We show that these

[2] This is easy to verify by calculating $\sum_{t=0}^{\infty}\delta_i^t = \frac{1}{1-\delta_i}$.

Table 4.1: Summary of notations.

Notation	Description
$\mathcal{N}$	Set of users.
$\mathcal{R}$	Set of resource types.
n	Number of users in the system.
m	Number of different resource types.
$\boldsymbol{\theta}_i^{(t)}$	User i's demands vector at instant t, $\boldsymbol{\theta}_i^{(t)} = \langle \theta_{i1}^{(t)}, \ldots, \theta_{im}^{(t)} \rangle$.
$\theta_{ir}^{(t)}$	User i's demand for resource r at instant t.
$\hat{\boldsymbol{\theta}}_i^{(t)}$	User i's *declared* demands vector at instant t, $\hat{\boldsymbol{\theta}}_i^{(t)} = \langle \hat{\theta}_{i1}^{(t)}, \ldots, \hat{\theta}_{im}^{(t)} \rangle$.
$\hat{\theta}_{ir}^{(t)}$	User i's *declared* demand for resource r at instant t.
$\boldsymbol{o}_i^{(t)}$	User i's allocation vector at instant t, $\boldsymbol{o}_i^{(t)} = \langle o_{i1}^{(t)}, \ldots, o_{im}^{(t)} \rangle$.
$o_{ir}^{(t)}$	User i's allocation for resource r at instant t.
$\boldsymbol{O}^{(t)}$	Matrix of all allocation vectors at instant t, $\boldsymbol{O}^{(t)} = \langle \boldsymbol{o}_1^{(t)}, \ldots, \boldsymbol{o}_n^{(t)} \rangle$.
$u_i^{(t)}(\boldsymbol{o}_i^{(t)})$	User i's utility function at instant t given an allocation $\boldsymbol{o}_i^{(t)}$.
$u_i^{[t,\infty)}$	User i's expected long-term utility at instant t.
δ_i	User i's discount factor.
δ	Parameter used in the calculation of commitments.
$\tilde{r}_i^{(t)}$	User i's dominant resource at instant t.
$\tilde{\boldsymbol{\theta}}_i^{(t)}$	User i's *normalized* demand vector at instant t, $\tilde{\boldsymbol{\theta}}_i^{(t)} = \langle \tilde{\theta}_{i1}^{(t)}, \ldots, \tilde{\theta}_{im}^{(t)} \rangle$.
$\tilde{\theta}_{ir}^{(t)}$	User i's *normalized* demand for resource r at instant t.
$c_{ir}^{(t)}$	User i's commitment for resource r at instant t.

properties alone are not enough to enforce fairness in the long run, requiring an alternative for the dynamic setting.

4.2.1 DRF Mechanism

Dominant Resource Fairness (DRF) [2] extends Max-Min Fairness (MMF) to the multi-resource setting. DRF calculates an allocation based on users' dominant resources (the most demanded resource for each user, relative to the total amount in the system). As we have normalized all the different kinds of resources to 1, we say $\tilde{r}_i^{(t)}$ is a dominant resource for user i at instant t, if

$$\tilde{r}_i^{(t)} \in \arg\max_{r \in \mathcal{R}} \theta_{ir}^{(t)}. \tag{4.3}$$

Given the dominant resource, we define the *normalized demand vector* for each user, in which the dominant resources become 1. The normalized demand vector for user i at instant t is denoted by $\tilde{\boldsymbol{\theta}}_i^{(t)} = \langle \tilde{\theta}_{i1}^{(t)}, \ldots, \tilde{\theta}_{im}^{(t)} \rangle$, where

$$\tilde{\theta}_{ir}^{(t)} = \frac{\hat{\theta}_{ir}^{(t)}}{\hat{\theta}_{i\tilde{r}_i^{(t)}}^{(t)}}, \forall i \in \mathcal{N}, r \in \mathcal{R}\,. \tag{4.4}$$

When users request an infinite number of tasks, DRF computes an allocation where each user receives an equal share of their dominant resource. For this particular case, DRF can be described using a simple linear program whose solution (x) is the share of dominant resource each user receives [81]:

$$\begin{aligned} \max_{x} \quad & x \\ \text{s.t.} \quad & \sum_{i \in \mathcal{N}} o_{ir}^{(t)} \leq 1, \, \forall r \in \mathcal{R}\,, \\ & o_{ir}^{(t)} = x \cdot \tilde{\theta}_{ir}^{(t)}\,. \end{aligned} \tag{4.5}$$

Intuitively, we increase x—and consequently the share of dominant resource for every user—until we achieve a bottleneck and no task can be allocated. Given x, the allocation for every user and resource can be calculated as $o_{ir}^{(t)} = x \cdot \tilde{\theta}_{ir}^{(t)}$.

4.2.2 Static Allocation Properties

In Chapter 2 we have introduced some desirable allocation properties. These properties have also been used in a variety of works [9, 81, 82] to ensure both fairness and efficiency in a static resource allocation. We now define these properties more formally using the model from §4.1. For the following definitions, consider a stage game happening at time t.

1. *Sharing Incentives* (SI). Users should be better off participating in the system than having a proportional and exclusive share of all the resources. Formally, we say that an allocation mechanism satisfies sharing incentives if for every user $i \in \mathcal{N}$, it outputs an allocation $\boldsymbol{o}_i^{(t)}$ such that, $u_i^{(t)}(\boldsymbol{o}_i^{(t)}) \geq u_i^{(t)}(\langle 1/n, \ldots, 1/n \rangle)$. This assumes users have the right to an equal share of all the resources. It is also possible to give users different weights, so that they have the right to a lower or higher share depending on their weights.

2. *Strategyproofness* (SP). Users should not benefit by misreporting their demands to the mechanism. Formally, if we denote the allocation returned by the mechanism when the user i reports her demands truthfully ($\hat{\boldsymbol{\theta}}_i^{(t)} = \boldsymbol{\theta}_i^{(t)}$) as $\boldsymbol{o}_i^{(t)}$ and when the user lies ($\hat{\boldsymbol{\theta}}_i^{(t)} \neq \boldsymbol{\theta}_i^{(t)}$) as $\boldsymbol{o}_i'^{(t)}$, then $u_i^{(t)}(\boldsymbol{o}_i^{(t)}) \geq u_i^{(t)}(\boldsymbol{o}_i'^{(t)})$.

3. *Pareto Optimality* (PO). The allocation should be optimal in the sense that if it can be changed to make a user's utility higher, it must make at least another user's utility lower (in other words the allocation cannot be Pareto dominated by another). Formally, an allocation mechanism is Pareto optimal if it returns an allocation $\boldsymbol{O}^{(t)}$ such that for any other feasible allocation $\boldsymbol{O}'^{(t)}$, if there is a user $i \in \mathcal{N}$ such that $u_i^{(t)}(\boldsymbol{o}_i'^{(t)}) > u_i^{(t)}(\boldsymbol{o}_i^{(t)})$ then there must be a user $j \in \mathcal{N}$ such that $u_j^{(t)}(\boldsymbol{o}_j'^{(t)}) < u_j^{(t)}(\boldsymbol{o}_j^{(t)})$.

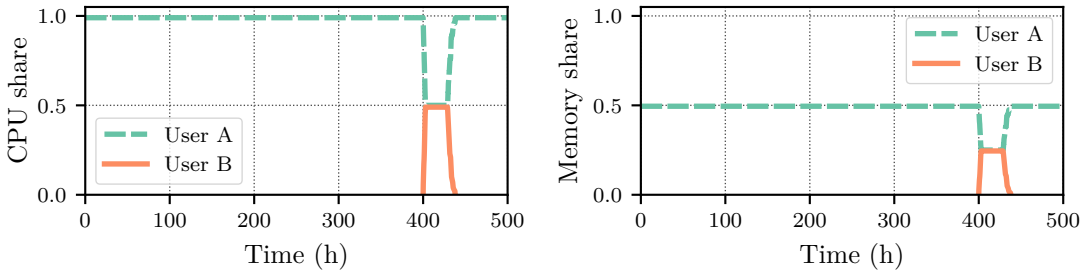


Figure 4.1: Unfairness in the long run. User B hardly uses the system but receives the same shares as user A.

In addition to the above properties, DRF also satisfies *envy-freeness*, which ensures that users never prefer another user's allocation to their own. Unfortunately satisfying both Pareto optimality and envy-freeness is impractical under indivisibilities [81]. Moreover, as we will see in the next subsection, while envy-freeness is usually desirable for static allocations, it does not ensure fairness in the dynamic setting.

4.2.3 Fairness in the Dynamic Setting

We now give motivation for an allocation policy that is fair in the long run. Previous works [2, 9] modeled users as having an infinite number of tasks with the same demand for each resource type. When this happens, only the share of resources each task needs is considered—time becomes irrelevant and the allocation is equivalent to a static one. In practice, however, while some users have workloads with repeated jobs, most users have quite dynamic workloads [35, 39].

To illustrate the importance of considering the past in an allocation, we present an example with users A, B and C sharing a system with a DRF scheduler (see Figure 4.1). There are two resources in the system, CPU and memory. User A's dominant resource is CPU and her normalized demand is $\langle 1, 0.5 \rangle$. User A is eager for resources and submits a huge amount of tasks. Nevertheless, the other users only use the system sporadically, with usage spikes. After user A is using the entire system for a while, user B has a spike with normalized demand $\langle 1, 0.5 \rangle$ as well. Even though user B never used her rightful share, the share she receives is the same as user A, *i.e.*, equal to 1/2. This demonstrates that the properties of fairness defined for a static allocation are not enough to enforce fairness in the long run. Satisfying sharing incentives guarantees that users will receive their rightful share but does not reward users for their lower usage. Envy-freeness assumes users are only aware of the present allocation and do not envy other users based on their past allocations.

4.2.4 Users' Commitments

To distinguish between users who constantly require more resources than their proportional share from users who only use the system sporadically, we introduce the concept of *commitment*. Commitment is a measure of users propensity to overuse their shares. The key intuition is that users who use more resources than their share, commit to use less if other users need. Users who overuse their shares for a short period of time should have lower commitment than users who constantly overuse. Also, users who overuse less resources should get lower commitment than users who overuse more resources (tuned by a parameter δ). Every user $i \in \mathcal{N}$ has a separate commitment for each resource $r \in \mathcal{R}$. We define commitment using an exponential moving average of overused resources. The user i's commitment for resource r at time t is given by:

$$c_{ir}^{(t)} = (1-\delta) \sum_{k=-\infty}^{t} \delta^{t-k} \bar{o}_{ir}^{(k)}, \tag{4.6}$$

where

$$\bar{o}_{ir}^{(k)} = \max \left\{ \left(o_{ir}^{(k)} - \frac{1}{n^{(k)}} \right), 0 \right\} . \tag{4.7}$$

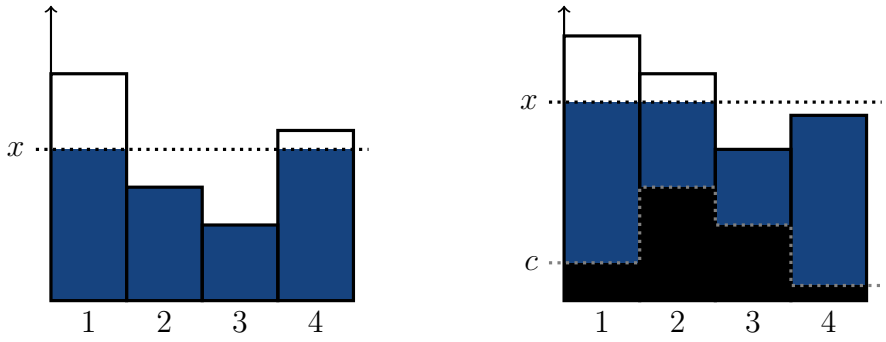
The term $n^{(k)}$ is the number of active and inactive users in the system at instant k. Therefore, the term $\bar{o}_{ir}^{(k)}$ represents how much user i overused her share for resource r on instant k. When this term is zero, the user did not overused her share. The more in the past users overused their share the less it influences their commitments.

4.3 Stateful Dominant Resource Fairness

In this section, we introduce *Stateful Dominant Resource Fairness* (SDRF), a generalization of DRF that improves fairness in the long run by enforcing users' commitments. First we develop a simpler version of SDRF for a single resource type. Then, we extend this version and obtain an optimization problem that yields an SDRF allocation. From this problem we proceed to prove that it satisfies the desired properties introduced in §4.2.

4.3.1 Stateful Max-Min Fairness

The intuition for SDRF is better understood if we first look at the single resource setting. Suppose we have a finite amount of a particular resource, *e.g.*, CPU cores, and we want to equally divide it among the users. The fairest way to divide it is to give an equal share of the resource for every user, *e.g.*, same number of CPU cores. Nonetheless, some users may not need their entire share, in that case it can be redistributed among the other users. This is the main principle behind *Max-Min*



(a) MMF. (b) SMMF.

Figure 4.2: Water-filling diagram for (a) MMF and (b) SMMF.

Fairness (MMF). One way to achieve MMF is to use a *water-filling algorithm* [36]. Water-filling progressively gives resources for every user until their demands are met. When a user demand is met, she stops receiving resources and the algorithm continues to give resources for the other users. Figure 4.2a shows the water-filling diagram for the MMF allocation. Each column (or tank) represents the total amount of resource each user demands. The resource is finite and progressively fills the tanks, until there is no more resource left. In the example, users 2 and 3 have their demands fulfilled while users 1 and 4 only have it partially fulfilled.

Even though MMF is fair for a static allocation, directly applying MMF to the dynamic setting causes the same problem as DRF—it does not consider the past and therefore cannot enforce fairness in the long run. To modify MMF to account for commitments, we introduce Stateful Max-Min Fairness (SMMF). The intuition behind SMMF is better illustrated by an example. "*If the equal share for the resource is 3 CPUs and the user has a commitment of 1 CPU, then the user should have the right to receive at least 2 CPUs.*" This notion can be directly implemented using the water-filling algorithm just by adding commitments as a "base for the tanks." Figure 4.2b shows the water-filling diagram for the SMMF allocation. Demands are the same as in Figure 4.2a, but now there is a base layer of arbitrary commitments c (black layer). Note how user 2 has a lower allocation than she would have without commitments, on the other hand, the demand for user 4 is now met.

Formally, the SMMF allocation can be defined using an optimization problem. Since SMMF allocates a single resource, the resources set becomes a singleton $\mathcal{R} = \{1\}$ and each user $i \in \mathcal{N}$ has a single allocation $o_{i1}^{(t)}$ at time t. The optimization problem maximizes x, the water level in Figure 4.2b, as long as there are resources

left in the system:

$$\begin{aligned} \max_{x} \quad & x \\ \text{s.t.} \quad & \sum_{i \in \mathcal{N}} o_{i1}^{(t)} \leq 1, \\ & o_{i1}^{(t)} = \max\left\{0, \min\left\{\hat{\theta}_{i1}^{(t)}, (x - c_{i1}^{(t)})\right\}\right\} . \end{aligned} \tag{4.8}$$

Given x, each user receives an allocation $o_{i1}^{(t)} = \max\{0, \min\{\hat{\theta}_{i1}^{(t)}, (x - c_{i1}^{(t)})\}\}$ which ensures that allocations are never above demands and remain nonnegative. When commitments are zero, SMMF is equivalent to MMF.

Having defined SMMF, we now generalize it to the multiple resources setting to finally obtain the SDRF mechanism.

4.3.2 SDRF Mechanism

SDRF generalizes SMMF similarly to the way DRF generalizes MMF to multiple resources. We use the same concept of dominant resource as DRF, defined in Eq. 4.3. Differently from DRF, though, we must deal with different commitments for different resources. We define the *dominant commitment* for a user i at time t as the user's largest commitment relative to the system total. As we have normalized all the resources to 1, the dominant commitment is simply the largest commitment for the user, *i.e.*,

$$\tilde{c}_i^{(t)} = \max_{r \in \mathcal{R}} \{c_{ir}^{(t)}\} . \tag{4.9}$$

Having defined the dominant commitment, we define SDRF using ideas from both DRF (Eq. 4.5) and SMMF (Eq. 4.8). Like DRF, SDRF increases the share of dominant resource for every user until a bottleneck is achieved. Like SMMF, users only start receiving resources when x is above their (dominant) commitment. SDRF is formally defined as:

$$\begin{aligned} \max_{x} \quad & x \\ \text{s.t.} \quad & \sum_{i \in \mathcal{N}} o_{ir}^{(t)} \leq 1, \, \forall r \in \mathcal{R} , \\ & o_{ir}^{(t)} = \max\left\{0, \min\left\{\hat{\theta}_{ir}^{(t)}, (x - \tilde{c}_i^{(t)}) \cdot \tilde{\theta}_{ir}^{(t)}\right\}\right\} . \end{aligned} \tag{4.10}$$

Recall $\tilde{\theta}_{ir}^{(t)}$ is the normalized demand for user i and resource r, defined in Eq. 4.4. From x, we may calculate the allocation for every user and resource by $o_{ir}^{(t)} = \max\{0, \min\{\hat{\theta}_{ir}^{(t)}, (x - \tilde{c}_i^{(t)}) \cdot \tilde{\theta}_{ir}^{(t)}\}\}$. In the next subsection we analyze the properties of SDRF that prove it behaves well in both the stage game and in the long run.

4.3.3 Analysis of SDRF Allocation Properties

We start our analysis of SDRF proving that it satisfies the desirable properties introduced in §4.2.2, namely: strategyproofness, Pareto optimality and sharing incentives. We defer the proofs of all propositions to §4.8.

First, we show that SDRF increases the share of dominant resource for every user until a resource runs out (the bottleneck resource). This is indicated in the following proposition.

Proposition 1 (Bottleneck). *The SDRF allocation obtained by solving Eq. 4.10 is such that all users have their demands fulfilled or there is a bottleneck resource. Formally,* $\boldsymbol{o}_i^{(t)} = \hat{\boldsymbol{\theta}}_i^{(t)}, \forall i \in \mathcal{N}$ *or* $\exists r \in \mathcal{R}$ *such that* $\sum_{i \in \mathcal{N}} o_{ir}^{(t)} = 1$.

Although simple, Proposition 1 is useful to demonstrate the following properties. One of the fundamental properties of DRF is strategyproofness. Without it, users may try to manipulate the system by, *e.g.*, faking their usage, which results in inefficiencies [2, 83]. Propositions 2 and 3 show SDRF is also strategyproof.

Proposition 2 (Strategyproofness in the Stage Game). *When users consider only the stage game utility (Eq. 4.1), the SDRF allocation obtained by solving Eq. 4.10 is strategyproof.*

Proposition 2 shows that when users consider only stage game utilities, SDRF is strategyproof. However, the fact that we consider past allocations may create new incentives for users to manipulate their declared demands. It may be possible that some users would not use the system when they actually need, hoping that this would improve their future allocations—this would also bring inefficiencies to the system. Fortunately, Proposition 3 shows that this is not possible.

Proposition 3 (Strategyproofness in the Repeated Game). *When users evaluate their utilities using the expected-long-term utility (Eq. 4.2), the SDRF allocation obtained by solving Eq. 4.10 is strategyproof, regardless of users' discount factors.*

The following two propositions demonstrate that SDRF is efficient. Proposition 4 shows that SDRF does not waste resources while Proposition 5 shows that the allocation is Pareto optimal, ensuring that it is not possible to increase a user's allocation without decreasing another.

Proposition 4 (Non-wastefulness). *The SDRF allocation* $\boldsymbol{O}^{(t)}$ *is such that, if there is a different allocation* $\boldsymbol{O'}^{(t)}$ *where* $o'^{(t)}_{ir} \leq o^{(t)}_{ir}, \forall i \in \mathcal{N}, r \in \mathcal{R}$ *and for a user* $i^* \in \mathcal{N}$ *and resource* $r^* \in \mathcal{R}$, $o'^{(t)}_{i^*r^*} < o^{(t)}_{i^*r^*}$, *then it must be that* $u^{(t)}_{i^*}(\boldsymbol{o}^{(t)}_{i^*}) > u^{(t)}_{i^*}(\boldsymbol{o'}^{(t)}_{i^*})$. *In other words, SDRF is non-wasteful.*

Proposition 5 (Pareto optimality). *The SDRF allocation obtained by solving Eq. 4.10 is Pareto optimal.*

The last property indicates that users are better off if they participate in the system. More specifically, it shows that users receive a utility at least as good as if they had access to $1/n$ of resources in the system.

Proposition 6 (Sharing incentives). *The SDRF allocation obtained by solving Eq. 4.10 satisfies sharing incentives.*

4.4 Implementation Using a Live Tree

In this section, we study how SDRF can be implemented in practice. We first consider the effect continuous time and indivisible tasks have in the model defined in §4.1. We then develop a *water-filling* algorithm to schedule tasks. Nevertheless, the algorithm requires users' priorities to be recalculated and sorted at every execution. To mitigate this problem we introduce *live tree*—a data structure that keeps elements sorted even with time-varying priorities—and show how it can be used to improve the SDRF scheduling algorithm.

4.4.1 Continuous Time

In the model defined in §4.1 we assume time progresses as a sequence of repeated games, suggesting a discrete time. The definition for commitment in Eq. 4.6 is compatible with this notion. In an actual system, however, tasks may arrive and finish at any instant, therefore we need an expression that allows us to compute commitment at continuous time. First we redefine Eq. 4.6 recursively using a difference equation [84],

$$c_{ir}^{(t)} = (1-\delta)\bar{o}_{ir}^{(t)} + \delta c_{ir}^{(t-\Delta t)} \tag{4.11}$$

where the commitment at time t can be calculated from commitment at time $t-\Delta t$. This assumes $\bar{o}_{ir}^{(t)}$ remains constant within the interval $(t-\Delta t, t]$. It turns out that Eq. 4.11 can be seen as an exponential smoothing and can be closely approximated in the continuous time [84], leading to the expression

$$\begin{aligned} c_{ir}^{(t)} &= (1-\mathring{\delta})\bar{o}_{ir}^{(t)} + \mathring{\delta} c_{ir}^{(t_0)} \\ \mathring{\delta} &= e^{-(t-t_0)/\tau}, \quad \tau = -\frac{\Delta t}{\ln(\delta)} \end{aligned} \tag{4.12}$$

where we may calculate $c_{ir}^{(t)}$ from any previous $c_{ir}^{(t_0)}$ as long as $o_{ir}^{(t)}$ remains constant from t_0 to t. Fortunately, o_{ir} only changes when a task for user i requiring resource r starts or finishes. In any other instant, o_{ir} remains constant, making Eq. 4.12 useful

in practice. This expression is analogous to the discrete version, using $\mathring{\delta}$ instead of δ. When $t - t_0 = \Delta t$, Eq. 4.12 becomes Eq. 4.11.

4.4.2 Indivisible Tasks

So far, we have assumed that tasks are arbitrarily divisible. This allowed us to give arbitrarily small amounts of resources to users. In practice, however, tasks are often not divisible [6, 34]. To schedule indivisible tasks we use the same approach as previous works [2, 9]—applying water-filling to tasks.

Algorithm 1 summarizes the task scheduling procedure. We define a set A of *active users* (users with at least one task waiting to be scheduled), and keep track of the total amount of resources allocated for every user. If the system is not full and if there is at least one user with a pending task, *i.e.*, $A \neq \emptyset$, we schedule the next task for the user with the lowest share of dominant resource compensated for commitments.

Algorithm 1 SDRF task scheduling

$A = \{1, \ldots, k\}$ ▷ set of active users
$\boldsymbol{o}_i = \langle o_{i1}, \ldots, o_{im} \rangle, \forall i \in A$ ▷ resources given to user i
$\boldsymbol{c}_i = \langle c_{i1}, \ldots, c_{im} \rangle, \forall i \in A$ ▷ commitments for user i
while $A \neq \emptyset$ **do**
  $i \leftarrow \arg\min_{i \in A} \left\{ \max_{r \in \mathcal{R}} \{o_{ir} + c_{ir}\} \right\}$ ▷ pick user
  $\forall r, \theta_{ir} \leftarrow$ demand for r in user i's next task
  if $\forall r, \left(\theta_{ir} + \sum_{j \in \mathcal{N}} o_{jr} \right) \leq 1$ **then**
    $\forall r, o_{ir} \leftarrow o_{ir} + \theta_{ir}$
    if no more pending tasks for user i **then**
      remove i from A
  else
    return ▷ the system is full

Whenever a task arrives or finishes, we rerun Algorithm 1 with updated set A, and vectors $\boldsymbol{o}_i$, $\boldsymbol{c}_i$, $\forall i \in A$. The smaller tasks are, the closer Algorithm 1 approximates Eq. 4.10.

Performance is a major concern in the design of a task scheduler. In peak hours, a scheduler may need to make hundreds of task placement decisions per second [35]. The most expensive part of Algorithm 1 is picking a user. While plain DRF can be implemented using a priority queue that stores the dominant resource share for every user[3], this is not possible for SDRF. In DRF, users' priorities only change when $\boldsymbol{o}_i$ changes, in SDRF users' commitments change at any instant and so do

[3]The DRF implementation on Mesos [6] uses a binary tree, an `std::set` from the C++ Standard Template Library.

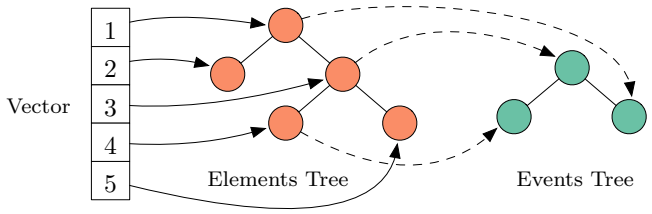


Figure 4.3: Illustration of a live tree with its data structures. Positions in the array link to elements in the tree. Some elements link to events in the events tree.

users' priorities. Recomputing priorities for every user and resource at every task scheduling decision would be too costly. The next subsection shows how to solve this problem.

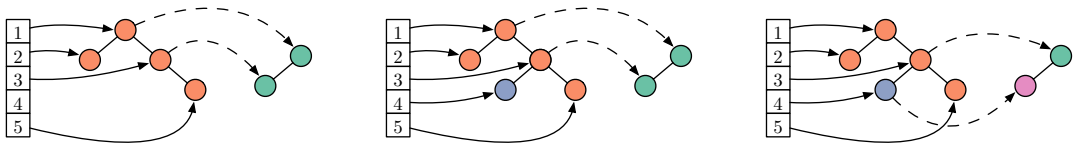
4.4.3 Live Tree

When scheduling tasks, we are not really interested in the specific value of commitments, but in which user has the *highest priority*. Live tree is a data structure that keeps elements with predictable time-varying priorities sorted. The key idea is to focus on position-change events, instead of element priorities. When priorities follow a continuous function, elements change position whenever their priorities intersect. A live tree always has a *current time* associated with it—for this current time, it guarantees that elements are sorted. When the current time is updated, instead of updating every element priority, we see if any position-change event happened from the last update to the current time.

Live tree can be seen as a combination of two red-black trees [85, 86] and an array (see Figure 4.3). We call one red-black tree *elements tree*, as it keeps elements sorted by priority, while the other is the *events tree*, as it tracks position-change events sorted by their time. The array is used for element lookups. For simplicity, we assume that each of the input elements has a distinct integer id that can be an index for the array[4]. Each position in the array has a pointer to an element in the elements tree (or NIL if there is no element for the given index). This allows us to retrieve elements by id in the tree in $O(1)$ time. If two neighboring elements in the elements tree are to change position in the future, the left element will have a pointer to a position-change event in the events tree.

We assume that priorities for all elements can be calculated using the same continuous function $p(t, \kappa)$ based on time t and in the element attribute κ. Every element has a different attribute that dictates how its priority changes over time. It may be a number, a vector or even a tuple. Our description does not depend on the definition of κ, in the next subsection we better define κ for our setting. It

[4]When elements do not have integer ids, or they are too sparse, the array may be replaced by a hash table and still present amortized $O(1)$ lookups.

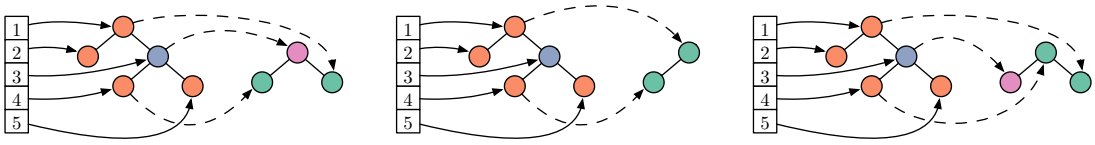


We will insert element 4.

Element 4 is between elements 1 and 3. Position 4 now points to Element 4.

Element 1 will not intersect with element 4. Element 4 will intersect with element 3.

Figure 4.4: Example of insertion: INSERT$(4, \kappa_4)$.



We will update the event for element 3.

We remove element 3's event from the events tree.

Elements 3 and 5 intersect before elements 4 and 3.

Figure 4.5: Example of event update: UPDATEEVENT(3).

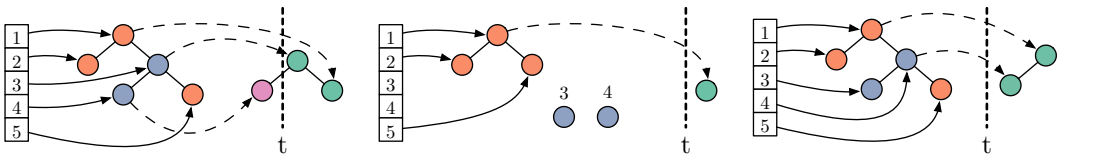
also helps to introduce an order notation, we say that element i precedes element j for an instant t, $i \prec_t j$, if $p(t, \kappa_i) < p(t, \kappa_j)$. The elements tree compares elements using $[\prec_t]$. This is useful since, whenever we insert a new element, it is compared to the others consistently with the time t. Live tree also needs a function to calculate priority intersections. We denote by $t_{\text{int}}(t, \kappa_i, \kappa_j)$ the function that calculates the priority intersection time based on two element attributes (κ_i, κ_j) and the time t.

We now briefly describe the basic operations of a live tree:[5]

INSERT(i, κ_i). Figure 4.4 shows an example of insertion. To insert an element i in the live tree, we first insert i in the elements tree. Since the elements tree compares elements using $[\prec_t]$, i will be placed in the correct position relative to time t. Once inserted, we set a pointer from position i in the array to the element in the tree. Then, we call UPDATEEVENT for i and for its predecessor in the tree. When i is the minimum element, we only call UPDATEEVENT for i. INSERT can be accomplished in $O(\log n)$ time.

UPDATEEVENT(i). If an element i will change position with its successor in the future, it must have a position-change event associated with it. Figure 4.5 shows an example of event update. To update an event we first check if the element i has an event in the events tree and remove it if so. Then, we check if i and its successor j will switch places in the future by calculating their priorities intersection $t_{\text{int}}(t, \kappa_i, \kappa_j)$. If t_{int} exists and is positive, we add an event for element i and time $t_{\text{int}} + t$ in the events tree. Then we add a pointer from element i in the elements tree

[5]Our implementation of SDRF and Live Tree is open source and is available at `https://github.com/hugombarreto/sdrf`

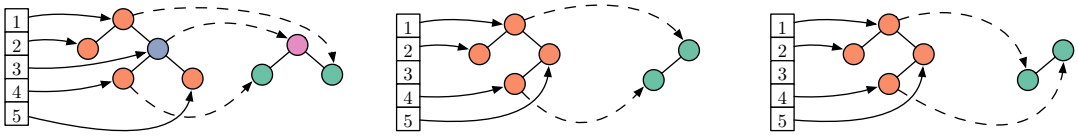


We will update the tree to time t. The first event happens before time t.

We remove elements 3 and 4 calling DELETE. Elements 1 and 5 intersect after time t.

We reinsert elements 3 and 4 calling INSERT.

Figure 4.6: Example of time update: UPDATE(t).



We will delete element 3.

We remove element 3 and its event. Element 4's event is now outdated.

Elements 4 and 5 intersect after 1 and 4. UPDATEEVENT(4) fixes the order.

Figure 4.7: Example of deletion: DELETE(3).

to the event in the events tree. When i is the maximum element, it has no successor and thus cannot have a position-change event (note this does not imply it cannot change position, as its predecessor can have an event). UPDATEEVENT can be done in $O(\log n)$ time.

UPDATE(t). Whenever the current time changes, we must update the tree. We assume that time progresses forward and live tree can only be updated to the future. Figure 4.6 shows an example of time update. To update the tree to a new time t, we look at all events that happen before t. If there is no event, *i.e.*, the first event in the events tree has time greater than t, then no element should change position and the tree is already updated, otherwise we must consider the events. We remove events from the events tree in order until the next event has time greater than t or the events tree becomes empty. For every removed event, we remove its correspondent element as well as its successor from the elements tree calling DELETE. Once we finish removing events, we reinsert each removed element calling INSERT. Since elements are compared using $[\prec_t]$, the reinsertion places elements in their correct position relative to time t. UPDATE can be accomplished in $O(n \log n)$ time. The worst case happens when every element must change position and therefore must be reinserted in the tree. In §4.5 we show that, for SDRF, the actual time is much smaller than the worst case.

DELETE(i). Figure 4.7 shows an example of deletion. To delete an element i, we first check position i in the array. From position i we get a pointer to the elements tree. If the element has an event, we get a pointer to its event as well. We then

remove the event from the events tree, the element from the elements tree and set NIL at position i in the array. If i was the minimum element, we are done, otherwise we must call UpdateEvent to the predecessor of i in the elements tree. Delete can be accomplished in $O(\log n)$ time.

Minimum/Maximum. The minimum (maximum) in the live tree is the minimum (maximum) in the elements tree. Minimum/Maximum can be accomplished in $O(1)$ time.

We omitted from our description corner cases, such as if an element being deleted does not exist, or if the element being inserted is already in the tree.

Live tree performance depends heavily on the priority function used and the frequency of Update calls. When elements change position often, Update has to process more events. Nevertheless, the higher the frequency of Update calls, the less events each call has to process. In §4.5 we evaluate how live tree performs when used to implement SDRF.

4.4.4 Live Tree Applied to SDRF

We now apply live tree to Algorithm 1. In Algorithm 1, we pick the user with the minimum value of $\max_{r\in\mathcal{R}}\{o_{ir} + c_{ir}\}$, therefore we use a live tree to sort users by this value. Using Eq. 4.12, we define the priority function p as

$$\begin{aligned} p(t,\kappa_i) &= \max_{r\in\mathcal{R}} \left\{ o_{ir} + (1-\mathring{\delta})\bar{o}_{ir} + \mathring{\delta} c_{ir}^{(t_i)} \right\} \\ \mathring{\delta} &= e^{-(t-t_i)/\tau} \end{aligned} \tag{4.13}$$

$\kappa_i = (t_i, \tau, o_{i1}, \ldots, o_{im}, \bar{o}_{i1}, \ldots, \bar{o}_{im}, c_{i1}^{(t_i)}, \ldots, c_{im}^{(t_i)})$, τ is defined as in Eq. 4.12 and is the same for all users. t_i is the time user i is inserted in the live tree.

To obtain the intersection function we calculate the time when any two arbitrary priorities intersect, *i.e.*, $p(t,\kappa_1) = p(t,\kappa_2)$. Since priorities are calculated from the maximum value of $o_{ir} + c_{ir}$, it is useful to define a set of all resource priority intersections, $\mathcal{I}_{ij}$. Whenever two resource priorities from users i and j intersect, the intersection will appear in this set. We derive the expression for the set $\mathcal{I}_{ij}$ calculating the time t that satisfies the equality:

$$o_{ir_1} + c_{ir_1}^{(t_0)} = o_{jr_2} + c_{jr_2}^{(t_0)}, \text{ where } t_0 = \max\{t_i, t_j\}\,.$$

Using Eq. 4.12,

$$o_{ir_1} + (1-\mathring{\delta})\bar{o}_{ir_1} + \mathring{\delta} c_{ir_1}^{(t_0)} = o_{jr_2} + (1-\mathring{\delta})\bar{o}_{jr_2} + \mathring{\delta} c_{jr_2}^{(t_0)}\,.$$

Isolating $\mathring{\delta}$,

$$\mathring{\delta} = \frac{\bar{o}_{ir_1} - \bar{o}_{jr_2} + o_{ir_1} - o_{jr_2}}{\bar{o}_{ir_1} - \bar{o}_{jr_2} + c_{jr_2}^{(t_0)} - c_{ir_1}^{(t_0)}}\,.$$

Replacing $\mathring{\delta}$ by $e^{-(t-t_0)/\tau}$,

$$e^{-(t-t_0)/\tau} = \frac{\bar{o}_{ir_1} - \bar{o}_{jr_2} + o_{ir_1} - o_{jr_2}}{\bar{o}_{ir_1} - \bar{o}_{jr_2} + c_{jr_2}^{(t_0)} - c_{ir_1}^{(t_0)}}\,.$$

Finally, isolating t,

$$t = t_0 + \tau \ln\left(\frac{\bar{o}_{ir_1} - \bar{o}_{jr_2} + c_{jr_2}^{(t_0)} - c_{ir_1}^{(t_0)}}{\bar{o}_{ir_1} - \bar{o}_{jr_2} + o_{ir_1} - o_{jr_2}}\right)\,. \tag{4.14}$$

Using Eq. 4.14 we formally define $\mathcal{I}_{ij}$ as

$$\mathcal{I}_{ij} = \left\{\tau \ln\left(\frac{\bar{o}_{ir_1} - \bar{o}_{jr_2} + c_{jr_2}^{(t_0)} - c_{ir_1}^{(t_0)}}{\bar{o}_{ir_1} - \bar{o}_{jr_2} + o_{ir_1} - o_{jr_2}}\right) \middle| (r_1, r_2) \in \mathcal{R}^2\right\},$$

where $t_0 = \max\{t_i, t_j\}$.

We define the intersection function getting the minimum intersection after the current time t,

$$t_{\text{int}}(t, \kappa_i, \kappa_j) = \min\left\{k + t_0 - t | k \in \mathcal{I}_{ij} \wedge k + t_0 > t\right\} \tag{4.15}$$

When there is no intersection after the time t, t_{int} does not exist and live tree will add no event. Note this intersection function may indicate intersections between resources that do not cause an intersection in priorities, *i.e.*, commitments may intersect without changing the dominant commitment. Although non-optimal, it performs correctly, as false events do not change the order in the tree. In the next section, we show how SDRF and live tree perform when scheduling tasks.

4.5 Simulation Results

In this section, we evaluate SDRF and live tree using trace-driven simulations based on Google cluster traces [35]. The traces contain information from workloads (from either Google services or engineers) running in a cluster over a month-long period. Workloads are submitted in the form of jobs, and each job may have multiple tasks. The traces contain events for every time a task is submitted, is scheduled or finishes. From these events we extract the CPU and memory demands as well as task submission and running times using them as input for our simulation. We remove tasks with 0 demand, as well as tasks that were evicted by the Google system, but

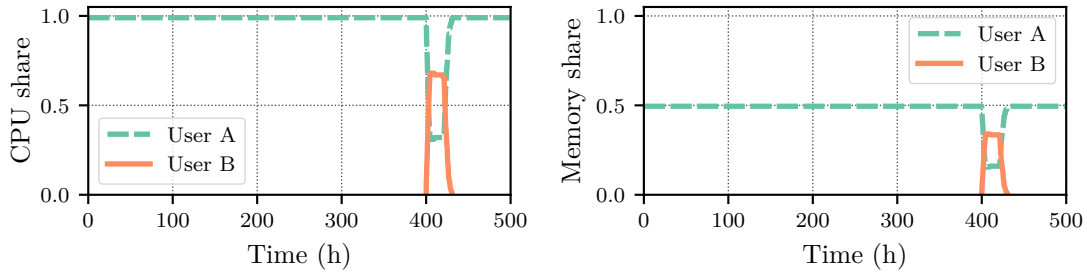


Figure 4.8: Same example as Figure 4.1 but using SDRF ($\delta = 1 - 10^{-6}$). Note how user B receives more resources and is able to complete her workload faster.

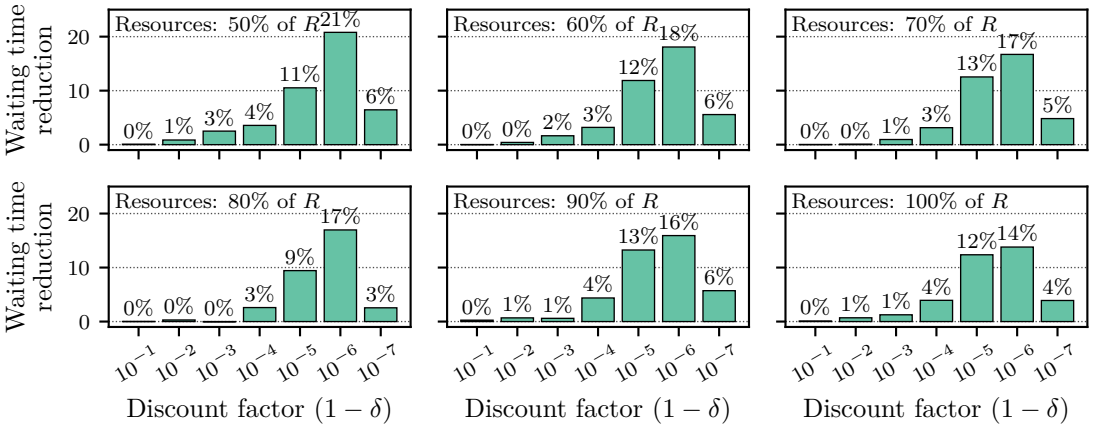


Figure 4.9: Mean wait time reduction for every user relative to DRF.

leave tasks that failed due to user errors. After that, we are left with around 32 million tasks from 627 users.

We run simulations for different values of δ and system overload. The values of δ are relative to a Δt of 1 second (see Eq. 4.12). We vary δ by making it exponentially closer to 1, *i.e.*, $\delta = 1 - 10^{-1}, \ldots, 1 - 10^{-7}$. This is equivalent to exponentially increasing τ from Eq. 4.12. To verify how SDRF performs under different levels of system load we also perform simulations for multiple values of total resources (*i.e.*, CPU and memory). We use the average system usage in the original trace, called hereinafter as R, as a baseline for our results. We then run simulations with the total amount of resources in the system varying from 50% to 100% of R, in steps of 10%.

Before running SDRF against the trace, we run it for the same example presented in §4.2 (Figure 4.1). Figure 4.8 shows how user B receives more resources (both CPU and memory) than user A and is able to complete her workload faster than using DRF. Since user A is constantly using the system, receiving less resources for a short period will have a low impact in her overall workload.

We now evaluate the simulation results. Figure 4.9 presents the mean waiting time reduction for every user under different values of δ and system load when compared to DRF. When δ is small enough, SDRF performs close to DRF. Also, for

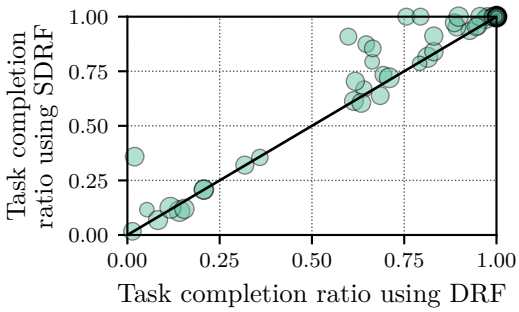


Figure 4.10: Task completion ratio using DRF and SDRF. Each bubble is a different user. The bubble's size is logarithmic to the number of tasks submitted by the user. Users above the $y = x$ are better with SDRF.

δ sufficiently close to 1, SDRF approaches DRF. This is justified inspecting Eq. 4.12: when δ is sufficiently close to 1, commitments never accumulate, alternatively, when δ is sufficiently close to 0, commitments are simply the last allocation, and therefore tasks are scheduled just like in DRF. The best waiting time reduction was observed for the discount factor $\delta = 1 - 10^{-6}$ for all levels of system load evaluated. Even though the advantage of SDRF is more evident when the system is overloaded, for $\delta = 1 - 10^{-6}$, SDRF consistently outperforms DRF by more than 10%.

We also investigate how the waiting time reduction affects the number of tasks each user is able to complete. We use the Google traces to compute the task completion ratio for every user (*i.e.*, the number of tasks completed divided by the number submitted) and compare it when running DRF and SDRF. Figure 4.10 shows the results for the simulation with $\delta = 1 - 10^{-6}$ and total resources 50% of R. Each bubble represents a different user: when above the black $y = x$ line, the user is able to complete more tasks under SDRF than under DRF. Most users perform better under SDRF, in fact, only 9 out of 627 users completed less tasks under SDRF. Also note that, even though these users completed less tasks, their task completion ratio had low impact. This happens because users that use the system in small bursts complete their workloads earlier and, consequently, have the opportunity to complete more tasks. On the other hand, users that use the system continually experience a low impact.

Next we evaluate how live tree performs under the same simulations. The theoretical worst case complexity for the update operation is $O(n \log n)$. This is driven by the maximum number of events an update may trigger, when there is no event, updates are performed in $O(1)$. In practice, however, the average number of events is much shorter. Figure 4.11 shows the number of live tree events that happened during the entire simulation period for every simulation. Each curve represents a different value of system load, 50% of R to 100% of R, from top to bottom. The number of events increases when the amount of resources in the system decreases.

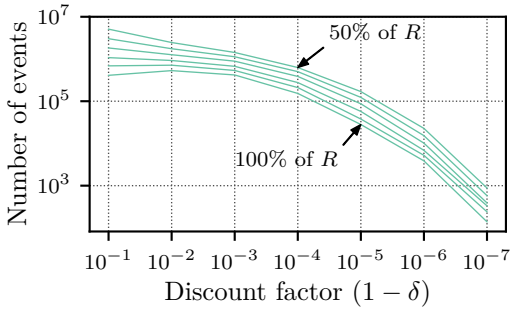


Figure 4.11: Live tree events for different values of discount factor and system resources (50% to 100% of R from top to bottom).

Also, the closer δ is to 1, the less events we observe. This makes sense, since commitments vary slower the closer δ is to 1. When $\delta = 1-10^{-6}$ and the total resources 50% of R, there is a total of 22,718 events, which is about 7 events for every 1,000 scheduled tasks. Since every task scheduling triggers one update, this indicates that updates happen fast for this scenario. Even for the worst scenario ($\delta = 1-10^{-1}$, 50% of R), the total number of events is 5,094,167, which is approximately 2 events for every 10 tasks. If live tree performed close to its theoretical worst case complexity, it would offer low advantage compared to sorting elements on every update. But with the number of updates observed, live tree operations will perform close to the ones of a red-black tree where weights are static.

4.6 Related Work

Fair resource allocation is a prevalent research topic, both in the computer science and economics fields. Nonetheless, focus is often given to the single resource setting. Ghodsi *et al.* [2] are the first to investigate the multi-resource setting under a shared computing perspective, proposing DRF. Dolev *et al.* [37] propose an alternative based on "bottleneck fairness." Nevertheless, the alternative is not strategyproof and is computationally expensive [83]. Gutman *et al.* [87] develop polynomial-time algorithms to compute both DRF and "bottleneck fairness" for non-discrete allocations. Joe-Wang *et al.* [8] extend the notion of fairness introduced by DRF to develop a framework that captures the fairness-efficiency tradeoff. However, they assume a cooperative environment and as such do not evaluate strategyproofness. Wang *et al.* [9] generalize DRF for a scenario with multiple heterogeneous servers, relaxing the sharing incentives restriction. Friedman *et al.* [88] also look at the allocation on multiple servers but provide a randomized solution that achieves sharing incentives. Another extension of DRF is proposed by Parkes *et al.* [81] to account for users with different weights and zero demands. Zarchy *et al.* [89] also investigate multi-resource allocation, but investigate what happens when the same application

may be developed differently to use different proportions of resource types. They propose a framework that allows users to submit multiple demands for the same application. Even though the aforementioned works consider the multi-resource setting, they ignore the dynamic nature of users' demands.

Bonald and Roberts [38] suggest Bottleneck Max Fairness (BMF), which also does not enforce strategyproofness, but improves resource utilization as compared to DRF. They consider dynamic demands in their analysis, arguing that for highly dynamic environments, such as networks, it is hard for users to manipulate the system. BMF convergence is proved in a later work [90]. Even though the analysis of BMF considers dynamic demands, the allocation itself considers only short term usage, ignoring fairness in the long run. Kash *et al.* [82] investigate a dynamic setting where users arrive and never leave, however, they also assume that demands remain constant. Friedman *et al.* [91] evaluate the scenario where multiple users arrive and leave the system. The focus, however, is on the fair division of resources as soon as the user arrives, limiting the number of task disruptions. There are also works that adapt DRF to packet processing [83, 92] and consider a recent past. Nevertheless, this is done to prevent limitations that arise when scheduling packets—in which resources must be shared in *time*—and not to ensure fairness and efficiency in the long run. Finally, other authors have focused on improving efficiency in the long run but not fairness [4, 93]. While some of these works consider users' dynamicity, they do not address fairness in the long run, which was our focus in this chapter.

Live Tree can be seen as an alternative implementation of a Kinetic Priority Queue [94]. Different from the classical implementations, however, Live Tree ensures strict upper bounds for priorities that follow an arbitrary continuous function. This happens because, in the classical implementations, elements are swapped—or rotated—in the presence of events [95]. Live Tree does not swap elements, instead, it removes pair of elements and reinsert them according to the update time t. By doing this, it ensures that every update has a cost of $O(\log n)$ for every changing pair of elements, regardless of the number of intersections that happen from the last update to the current. This is particularly useful for SDRF, since the function that calculates weights is not trivial (Eq. 4.15).

4.7 Conclusion

In this chapter, we introduced SDRF, an extension of DRF that enforces fairness in the long run. SDRF looks at past allocations and benefits users with lower average usage. We showed that SDRF satisfies the fundamental properties of DRF while enforcing fairness in the long run. To efficiently implement SDRF, we introduced live tree, a general-purpose data structure that keeps elements with predictable

time-varying priorities sorted. We simulated SDRF using Google cluster traces for a month-long period. Results have shown that under SDRF, users with low utilization can complete their workloads faster. Meanwhile, users with high utilization suffer a low impact in their overall workload. We also used the simulations to evaluate live tree performance, concluding that SDRF can be implemented efficiently.

There are different future investigation directions. First, we believe live tree may benefit other applications, *e.g.*, Dijkstra's algorithm applied to graphs with time-variable weights. Second, SDRF can be extended to cover other applications, *e.g.*, collaborative clouds [96]. Moreover, although we mentioned the possibility of using weights for users, we did not evaluate it formally. Another possibility is the use of a different function to measure commitments.

4.8 Deferred Proofs

In this section, we prove the propositions stated in §4.3.3. Before continuing, we introduce a simple lemma. This lemma states that the allocation a user gets for a certain resource will always be the normalized demand for this resource multiplied by the allocation for the dominant resource. For example, if the normalized demand vector for a user is $\langle 0.5, 1\rangle$ and the allocation for the dominant resource is 0.2, then, the allocation would be $\langle 0.1, 0.2\rangle$.

Lemma 1. *Given an SDRF allocation $\boldsymbol{O}^{(t)}$, obtained by solving Eq. 4.10, the allocation user i receives for resource r is such that $o_{ir}^{(t)} = \tilde{\theta}_{ir}^{(t)} o_{i\tilde{r}_i^{(t)}}^{(t)}, \forall i \in \mathcal{N}, r \in \mathcal{R}$.*

Proof. The proof is straightforward. From Eq. 4.10 we know that $o_{ir}^{(t)} \in [0, \hat{\theta}_{ir}^{(t)}]$.

- When $o_{ir}^{(t)} \in (0, \hat{\theta}_{ir}^{(t)})$:

$$o_{ir}^{(t)} = (x - \tilde{c}_i^{(t)}) \cdot \tilde{\theta}_{ir}^{(t)} \,. \tag{4.16}$$

  By replacing $x - \tilde{c}_i^{(t)} = o_{ir}^{(t)} / \tilde{\theta}_{ir}^{(t)}$ in Eq. 4.10, and as long as $o_{i\tilde{r}_i^{(t)}} \in (0, \hat{\theta}_{i\tilde{r}_i^{(t)}}^{(t)})$, we get to

$$o_{i\tilde{r}_i^{(t)}}^{(t)} = o_{ir}^{(t)} / \tilde{\theta}_{ir}^{(t)} \,,$$

  therefore,

$$o_{ir}^{(t)} = \tilde{\theta}_{ir}^{(t)} o_{i\tilde{r}_i^{(t)}}^{(t)}. \tag{4.17}$$

  Thus we just need to prove that $o_{i\tilde{r}_i^{(t)}}^{(t)} \in (0, \hat{\theta}_{i\tilde{r}_i^{(t)}}^{(t)})$. In fact, $o_{i\tilde{r}_i^{(t)}}^{(t)} > 0$, since we are considering $o_{ir}^{(t)} > 0$ and by definition $\hat{\theta}_{i\tilde{r}_i^{(t)}}^{(t)} > 0$. Verifying the upper bound is also straightforward. We depart from $o_{ir}^{(t)} < \hat{\theta}_{ir}^{(t)}$ and use the definition in Eq. 4.4,

$$\tilde{\theta}_{ir}^{(t)} \hat{\theta}_{i\tilde{r}_i^{(t)}}^{(t)} > o_{ir}^{(t)}$$

$$\hat{\theta}^{(t)}_{i\tilde{r}_i^{(t)}} > o^{(t)}_{ir} / \tilde{\theta}^{(t)}_{ir} = o^{(t)}_{i\tilde{r}_i^{(t)}} .$$

- When $o^{(t)}_{ir} = 0$:

$$(x - \tilde{c}_i^{(t)}) \cdot \tilde{\theta}^{(t)}_{ir} \leq 0$$

but $\tilde{\theta}^{(t)}_{ir} > 0$, therefore

$$x - \tilde{c}_i^{(t)} \leq 0 .$$

Making $x - \tilde{c}_i^{(t)} \leq 0$ in Eq. 4.10 we get to

$$o^{(t)}_{i\tilde{r}_i^{(t)}} = 0,$$

therefore Eq. 4.17 still holds.

- When $o^{(t)}_{ir} = \hat{\theta}^{(t)}_{ir}$:

$$(x - \tilde{c}_i^{(t)}) \cdot \tilde{\theta}^{(t)}_{ir} \geq \hat{\theta}^{(t)}_{ir} .$$

Using the definition in Eq. 4.4,

$$x - \tilde{c}_i^{(t)} \geq \hat{\theta}^{(t)}_{i\tilde{r}_i^{(t)}}$$

Making $x - \tilde{c}_i^{(t)} \geq \hat{\theta}^{(t)}_{i\tilde{r}_i^{(t)}}$ in Eq. 4.10 we get to

$$\begin{aligned} o^{(t)}_{i\tilde{r}_i^{(t)}} &= \hat{\theta}^{(t)}_{i\tilde{r}_i^{(t)}} \\ &= \hat{\theta}^{(t)}_{ir} / \tilde{\theta}^{(t)}_{ir} , \end{aligned}$$

which is equivalent to Eq. 4.17 when $o^{(t)}_{ir} = \hat{\theta}^{(t)}_{ir}$, concluding the proof.

□

Corollary 1 (of Lemma 1). *For a given user $i \in \mathcal{N}$, the allocation-demand ratio remains constant for every resource $r \in \mathcal{R}$, i.e., $\min_{r \in \mathcal{R}}\{o^{(t)}_{ir} / \hat{\theta}^{(t)}_{ir}\} = o^{(t)}_{ir} / \hat{\theta}^{(t)}_{ir}, \forall i \in \mathcal{N}, r \in \mathcal{R}$.*

Proof. From Lemma 1 and from Eq. 4.4,

$$o^{(t)}_{ir} / \hat{\theta}^{(t)}_{ir} = o^{(t)}_{i\tilde{r}_i^{(t)}} / \hat{\theta}^{(t)}_{i\tilde{r}_i^{(t)}}, \forall i \in \mathcal{N}, r \in \mathcal{R}.$$

Therefore,

$$\min_{r \in \mathcal{R}}\{o^{(t)}_{ir} / \hat{\theta}^{(t)}_{ir}\} = o^{(t)}_{ir} / \hat{\theta}^{(t)}_{ir}, \forall i \in \mathcal{N}, r \in \mathcal{R}.$$

□

Corollary 1 implies that for a user to improve her utility, she must increase the

allocation for every resource, otherwise the minimum allocation-demand ratio would not change.

We now turn to the proof of Proposition 1. It shows that if a user did not have her demand fulfilled, there must be at least one resource that is fully utilized—a bottleneck resource.

Proposition 1 (Bottleneck). *The SDRF allocation obtained by solving Eq. 4.10 is such that all users have their demands fulfilled or there is a bottleneck resource. Formally,* $\boldsymbol{o}_i^{(t)} = \hat{\boldsymbol{\theta}}_i^{(t)}, \forall i \in \mathcal{N}$ *or* $\exists r \in \mathcal{R}$ *such that* $\sum_{i \in \mathcal{N}} o_{ir}^{(t)} = 1$.

Proof. Assume, by way of contradiction, that we can obtain an allocation $\boldsymbol{O}^{(t)}$ from Eq. 4.10 where $\exists i \in \mathcal{N}$ such that $\boldsymbol{o}_i^{(t)} \neq \hat{\boldsymbol{\theta}}_i^{(t)}$ and $\sum_{i \in \mathcal{N}} o_{ir}^{(t)} \neq 1, \forall r \in \mathcal{R}$.

First, from the problem restrictions in Eq. 4.10, we know that $o_{ir}^{(t)} \leq \hat{\theta}_{ir}^{(t)}, \forall i \in \mathcal{N}, r \in \mathcal{R}$ and $\sum_{i \in \mathcal{N}} o_{ir}^{(t)} \leq 1, \forall r \in \mathcal{R}$. Thus,

$$\sum_{i \in \mathcal{N}} o_{ir}^{(t)} < 1, \forall r \in \mathcal{R} \text{ and} \tag{4.18}$$

$$\exists i \in \mathcal{N}, r \in \mathcal{R} \text{ such that } o_{ir}^{(t)} < \hat{\theta}_{ir}^{(t)}. \tag{4.19}$$

We now verify if we can propose a different solution to the problem:

$$x' = x + \epsilon, \epsilon \in \mathbb{R}_+.$$

If we can find a positive ϵ in which x' satisfies the problem constraints this is a contradiction (since $x' > x$ it could have been a solution to Eq. 4.10 instead of x). We denote by $\boldsymbol{O'}^{(t)}$ the allocation found using x'. If we can obtain a positive ϵ that satisfies the constraints, the following expression should hold,

$$\sum_{i \in \mathcal{N}} o'^{(t)}_{ir} \leq \sum_{i \in \mathcal{N}} \left(o_{ir}^{(t)} + \epsilon \, \tilde{\theta}_{ir}^{(t)} \right) \leq 1, \forall r \in \mathcal{R}\,.$$

Therefore,

$$\epsilon \leq \min_{r \in \mathcal{R}} \left\{ \frac{1 - \sum_{i \in \mathcal{N}} o_{ir}^{(t)}}{\sum_{i \in \mathcal{N}} \tilde{\theta}_{ir}^{(t)}} \right\}.$$

Since we want $\epsilon > 0$, we need to prove that

$$\min_{r \in \mathcal{R}} \left\{ \frac{1 - \sum_{i \in \mathcal{N}} o_{ir}^{(t)}}{\sum_{i \in \mathcal{N}} \tilde{\theta}_{ir}^{(t)}} \right\} > 0.$$

From Eq. 4.18, we know that

$$1 - \sum_{i \in \mathcal{N}} o_{ir}^{(t)} > 0, \forall r \in \mathcal{R}.$$

Therefore we need to verify if

$$\sum_{i \in \mathcal{N}} \tilde{\theta}_{ir}^{(t)} > 0, \forall r \in \mathcal{R},$$

which is also true since $\theta_{ir}^{(t)} > 0, \forall i \in \mathcal{N}, r \in \mathcal{R}$. Thus we can find an $\epsilon > 0$, making $x' > x$, which is a contradiction. □

Proposition 2 (Strategyproofness in the Stage Game). *When users consider only the stage game utility (Eq. 4.1), the SDRF allocation obtained by solving Eq. 4.10 is strategyproof.*

Proof. Take an arbitrary user $j \in \mathcal{N}$, we denote the allocation returned by the mechanism for the user j when $\hat{\boldsymbol{\theta}}_j = \boldsymbol{\theta}_j$ by $\boldsymbol{o}_j$, and when $\hat{\boldsymbol{\theta}}_j \neq \boldsymbol{\theta}_j$ by $\boldsymbol{o}'_j$. We must prove that

$$u_j(\boldsymbol{o}_j) \geq u_j(\boldsymbol{o}'_j)\,. \tag{4.20}$$

We check the following three cases.

- When $u_j(\boldsymbol{o}_j) = 1$: the utility cannot be improved and Eq. 4.20 trivially holds.
- When $u_j(\boldsymbol{o}_j) = 0$: from Corollary 1, $o_{jr} = 0, \forall r \in \mathcal{R}$. Therefore $x - \tilde{c}_j \leq 0$, and the allocation is zero for any resource, independently from $\hat{\boldsymbol{\theta}}_j$, which makes Eq. 4.20 true.
- When $0 < u_j(\boldsymbol{o}_j) < 1$: for this case, allocations are not limited by $\hat{\boldsymbol{\theta}}_j$. We may simplify the expression for the allocation in Eq. 4.10

  $$o_{jr} = (x - \tilde{c}_j) \cdot \tilde{\theta}_{jr}\,.$$

  From Proposition 1 there is a bottleneck resource, and we denote it as r^*. From Corollary 1 users must increase the allocation for all resources in order to improve their utilities. Therefore, if we can prove that it is impossible to have an alternative demand vector that increases the share of both the bottleneck resource and the dominant resource, we are done.

  We denote by $\tilde{\boldsymbol{\theta}}_j$ the truthful normalized demand vector for user j, and by $\tilde{\boldsymbol{\theta}}'_j$, the misreported normalized demand for user j.

  Increasing the share of dominant resource: The share of dominant resource user j receives when declaring the truth is given by:

  $$o_{j\tilde{r}_j} = (x - \tilde{c}_j) \cdot \tilde{\theta}_{j\tilde{r}_j} = x - \tilde{c}_j.$$

Therefore, to increase the share of dominant resource for user j (*i.e.*, make $o'_{j\tilde{r}_j} > o_{j\tilde{r}_j}$), we must make $x' > x$, where x' is the solution to Eq. 4.10 when user j declares $\tilde{\boldsymbol{\theta}}'_j$ instead of $\tilde{\boldsymbol{\theta}}_j$.

Increasing the share of bottleneck resource: Since r^* is a bottleneck resource, the following holds:

$$o_{jr^*} + \sum_{i \in \mathcal{N} \setminus \{j\}} o_{ir^*} = 1$$

Therefore, to make $o'_{jr^*} > o_{jr^*}$, we must decrease the sum of allocations of all users but j:

$$\sum_{i \in \mathcal{N} \setminus \{j\}} o'_{ir^*} < \sum_{i \in \mathcal{N} \setminus \{j\}} o_{ir^*}$$

and, consequently, since r^* is a bottleneck resource,

$$\sum_{i \in \mathcal{N} \setminus \{j\}} (x' - \tilde{c}_i)\tilde{\theta}_{ir^*} < \sum_{i \in \mathcal{N} \setminus \{j\}} (x - \tilde{c}_i)\tilde{\theta}_{ir^*}$$

$$\sum_{i \in \mathcal{N} \setminus \{j\}} (x' - \tilde{c}_i) < \sum_{i \in \mathcal{N} \setminus \{j\}} (x - \tilde{c}_i)$$

$$x' < x$$

Since increasing the share of *dominant* resource requires $x' > x$, while increasing the share of *bottleneck* resource requires $x' < x$, it is not possible to increase both shares, which concludes the proof.

□

Proposition 3 (Strategyproofness in the Repeated Game). *When users evaluate their utilities using the expected-long-term utility (Eq. 4.2), the SDRF allocation obtained by solving Eq. 4.10 is strategyproof, regardless of users' discount factors.*

Proof. Take an arbitrary user $i \in \mathcal{N}$, assuming user i discounts her utility using δ_i and the allocation mechanism calculates commitments using δ. Without loss of generality, we represent the expected-long-term utility for time $t = 0$ by

$$u_i^{[0,\infty)} = \mathbb{E}_{u_i}\left[(1 - \delta_i)\sum_{k=0}^{\infty} \delta_i^k u_i^{(k)}(\boldsymbol{o}_i^{(k)})\right]. \tag{4.21}$$

Since manipulating the stage game is not possible, the only hope users may have of improving their expected-long-term utility is by reducing their commitments. To reduce their commitments, users may declare a lower demand. We will show that any marginal gain the user may get, does not compensate her loss in the stage game.

If a user i declares a demand $\hat{\theta}_{ir}^{(0)} = \theta_{ir}^{(0)} - \epsilon$, with $0 < \epsilon < \theta_{ir}^{(0)}$, for a resource r, in the best scenario, this will make user i's commitment

$$c_{ir}^{\prime(k)} = c_{ir}^{(k)} - (1-\delta)\delta^k\epsilon, \tag{4.22}$$

where $c_{ir}^{\prime(k)}$ is the new commitment user i gets by declaring $\hat{\theta}_{ir}^{(0)}$. From this commitment, the maximum possible improvement in the long-term utility is

$$\begin{aligned}\bar{u}_i^{[0,\infty)} &= -(1-\delta_i)\epsilon + (1-\delta_i)\sum_{k=1}^{\infty}\delta_i^k(1-\delta)\delta^k\epsilon \\ &= -\epsilon(1-\delta_i) + \epsilon(1-\delta_i)(1-\delta)\sum_{k=1}^{\infty}(\delta_i\cdot\delta)^k.\end{aligned}$$

Then, replacing the infinite series,

$$\begin{aligned}\bar{u}_i^{[0,\infty)} &= -\epsilon(1-\delta_i) + \epsilon(1-\delta_i)(1-\delta)\left(\frac{1}{1-\delta_i\cdot\delta} - 1\right) \\ &= \epsilon(1-\delta_i)\left(\frac{(1-\delta)}{1-\delta_i\cdot\delta} - (1-\delta) - 1\right).\end{aligned}$$

Inspecting the expression we verify that, for $0 \leq \delta < 1$ and $0 \leq \delta_i < 1$,

$$\frac{(1-\delta)}{1-\delta_i\cdot\delta} < (1-\delta) + 1$$

and therefore,

$$\bar{u}_i^{[0,\infty)} < 0\,.$$

Thus, any positive decrement ϵ in the declared demands cannot possibly improve the expected-long-term utility, independently from users discount factors. □

Proposition 4 (Non-wastefulness). *The SDRF allocation $\boldsymbol{O}^{(t)}$ is such that, if there is a different allocation $\boldsymbol{O'}^{(t)}$ where $o_{ir}^{\prime(t)} \leq o_{ir}^{(t)}, \forall i \in \mathcal{N}, r \in \mathcal{R}$ and for a user $i^* \in \mathcal{N}$ and resource $r^* \in \mathcal{R}$, $o_{i^*r^*}^{\prime(t)} < o_{i^*r^*}^{(t)}$, then it must be that $u_{i^*}^{(t)}(\boldsymbol{o}_{i^*}^{(t)}) > u_{i^*}^{(t)}(\boldsymbol{o}_{i^*}^{\prime(t)})$. In other words, SDRF is non-wasteful.*

Proof. Assuming truthful demands, using Corollary 1 and the definition in Eq. 4.1, we have

$$u_i^{(t)}(\boldsymbol{o}_i^{(t)}) = o_{ir}^{(t)}/\theta_{ir}^{(t)}, \forall i \in \mathcal{N}, r \in \mathcal{R}\,.$$

Using the definition of $\boldsymbol{O'}^{(t)}$,

$$o_{i^*r^*}^{\prime(t)}/\theta_{i^*r^*}^{(t)} < o_{i^*r^*}^{(t)}/\theta_{i^*r^*}^{(t)} = u_{i^*}^{(t)}(\boldsymbol{o}_{i^*}^{(t)}).$$

However,

$$u_{i^*}^{(t)}(\boldsymbol{o}_{i^*}^{\prime(t)}) = \min\left\{\min_{r\in\mathcal{R}}\{o'^{(t)}_{i^*r}/\theta_{i^*r}^{(t)}\}, 1\right\}$$
$$\leq o'^{(t)}_{i^*r^*}/\theta_{i^*r^*}^{(t)}\,.$$

Therefore,

$$u_{i^*}^{(t)}(\boldsymbol{o}_{i^*}^{\prime(t)}) < u_{i^*}^{(t)}(\boldsymbol{o}_{i^*}^{(t)}),$$

concluding the proof. □

Proposition 5 (Pareto optimality). *The SDRF allocation obtained by solving Eq. 4.10 is Pareto optimal.*

Proof. The proof is a direct consequence of Propositions 1 and 4. For the sake of contradiction, assume SDRF is not Pareto optimal, then, from the allocation $\boldsymbol{O}^{(t)}$ obtained by solving Eq. 4.10 we can get another allocation $\boldsymbol{O}'^{(t)}$ that makes a user $i^* \in \mathcal{N}$ strictly better while making everybody's utility at least as good, *i.e.*,

$$u_{i^*}^{(t)}(\boldsymbol{o}_{i^*}^{\prime(t)}) > u_{i^*}^{(t)}(\boldsymbol{o}_{i^*}^{(t)}) \text{ and} \tag{4.23}$$
$$\forall i \in \mathcal{N}, u_i^{(t)}(\boldsymbol{o}_i^{\prime(t)}) \geq u_i^{(t)}(\boldsymbol{o}_i^{(t)}). \tag{4.24}$$

From Proposition 4, if $\exists i \in \mathcal{N}, r \in \mathcal{R}$ such that $o_{ir}^{\prime(t)} < o_{ir}^{(t)}$, then $u_i^{(t)}(\boldsymbol{o}_i^{\prime(t)}) < u_i^{(t)}(\boldsymbol{o}_i^{(t)})$. Therefore we may rewrite Eq. 4.24 as

$$o_{ir}^{\prime(t)} \geq o_{ir}^{(t)}, \forall i \in \mathcal{N}, r \in \mathcal{R}. \tag{4.25}$$

Also, from Corollary 1, we know that we must increase the allocation for all resources in order to increase the utility of a user, *i.e.*, for the user i^*,

$$o_{i^*r}^{\prime(t)} > o_{i^*r}^{(t)}, \forall r \in \mathcal{R}.$$

From Proposition 1, either

$$\boldsymbol{o}_i^{(t)} = \hat{\boldsymbol{\theta}}_i^{(t)} \text{ or} \tag{4.26}$$
$$\exists r^* \in \mathcal{R} \text{ such that } \sum_{i\in\mathcal{N}} o_{ir^*}^{(t)} = 1\,. \tag{4.27}$$

If Eq. 4.26 is true, then $u_i^{(t)}(\boldsymbol{o}_i^{(t)}) = 1, \forall i \in \mathcal{N}$ and Eq. 4.23 cannot be true. Therefore Eq. 4.27 must be true. But since Eq. 4.27 is true, when we make $o_{i^*r^*}^{\prime(t)} > o_{i^*r^*}^{(t)}$, we must decrease the allocation of another user, contradicting Eq. 4.25. □

Proposition 6 (Sharing incentives). *The SDRF allocation obtained by solving Eq. 4.10 satisfies sharing incentives.*

Proof. To prove sharing incentives, it is sufficient to show that there is a strategy, user i may follow, which makes her utility at least as good as $u_i^{(t)}(\langle 1/n, \ldots, 1/n \rangle)$, regardless of other users actions. We show that the strategy "always declare $\hat{\theta}_i^{(t)} = \langle 1/n, \ldots, 1/n \rangle$" guarantees that $o_i^{(t)} = \langle 1/n, \ldots, 1/n \rangle$ for every instant t.

From Eq. 4.10, $o_{ir}^{(t)} \leq \hat{\theta}_{ir}^{(t)}$ for every $r \in \mathcal{R}$, therefore, $o_{ir}^{(t)} \leq 1/n$. From Eq. 4.6, this makes $c_{ir}^{(t)} = 0$ for every resource r and instant t, also making $\tilde{c}_i^{(t)} = 0$ (from Eq. 4.9). Since $\tilde{c}_i^{(t)} = 0$, the allocation received by i will be $o_{ir}^{(t)} = \min\{x, 1/n\}$ for every resource r and instant t. Nevertheless, $x \geq 1/n$, which makes $o_{ir}^{(t)} = 1/n$, concluding the proof. □

Chapter 5

Conclusions and the Future of Networks and Datacenters

In this thesis, we have looked at ways of improving efficiency and fairness on two distinct shared systems: software middleboxes and datacenters. In this final chapter we summarize our contributions and propose directions for future work.

In Chapter 3 we took inspiration from modern datacenter networks and hypothesized that multi-core software middleboxes could also benefit from load-balancing packets at a finer granularity than flows. To validate this hypothesis we designed and implemented Sprayer. Sprayer not only configures the NIC to send packets from the same flow to multiple cores, but also provides abstractions for handling flow state in such context. We verified that, for the number of concurrent flows typical of real workloads, Sprayer improves fairness and TCP throughput even though it reorders packets.

In Chapter 4 we departed from the observation that current task schedulers do not ensure fairness in the long run. This ends up benefiting users with long-running jobs more than users that use the system sporadically. With SDRF, we showed that it is possible to allocate tasks more efficiently and improve long-term fairness by considering past allocations. To efficiently implement SDRF we also proposed live tree, a new data structure that keeps elements with predictable time-varying priorities sorted. We proved SDRF keeps the same fundamental properties of DRF and used trace-driven simulations to show that it reduces the waiting time for low-demand users, while having a low impact on high-demand users.

Before finishing, we highlight trends in hardware and applications that are likely to affect both networks and schedulers. Moreover, we point to problems and opportunities that are consequence of these trends.

5.1 Domain-Specific Architectures

In the last decades, CPUs obtained orders-of-magnitude performance improvements from architecture innovations, such as out-of-order execution and caches, as well as from Moore's Law.[1] Moore's Law, however, is coming to an end, and in the last few years hardware architects have been struggling to achieve even small performance improvements [97]. In response to this limitation, chip designers now widely believe that to continue to increase performance, while still providing programmability, there must be a move towards Domain-Specific Architectures (DSAs) [97]. DSAs are specialized chips that are less flexible than general-purpose CPUs but more efficient in their domains. Note that DSAs are different from strict ASICs, since DSAs serve not one, but a domain of applications. A recent example of DSA is Google's Tensor Processing Unit (TPU) that runs deep neural networks 15 to 30 times faster than contemporary CPUs [98].

If the DSA trend continues, datacenters are likely to have multiple types of specialized chips to accommodate different application domains. In such a setting, schedulers must be able to decide which tasks will use specialized chips and which will run on general-purpose CPUs. As is the case today with GPUs, some tasks will benefit more from running on DSAs than others. Designing a scheduler that takes this into account while ensuring the properties listed in §2.2.1 is an interesting research direction.

DSAs are also starting to appear in networks, with a movement towards programmable NICs [73, 74] and switches [99–102]. Programmable NICs and switches not only speedup the deployment of new protocols but also open new avenues for improving congestion control and packet scheduling. In §3.5 we have discussed a few ways in which programmable NICs could be used to improve Sprayer, but we expect to see many other applications.

5.2 Decentralized Control and Computation

Recently, there has been a surge of applications with decentralized control and computation. For instance, fog computing—a move of the cloud computing paradigm to the edge of the network—is gaining traction as a way of fulfilling the latency, mobility and scalability requirements of the Internet of Things [103, 104]. A scheduler for the fog must also take these requirements into consideration.

Another increasingly popular set of applications with decentralized control and computation are cryptocurrencies. Cryptocurrencies make use of a distributed ledger to record transactions and avoid double spending. One of the side effects of the

[1] Moore's law states that the number of transistors per chip doubles every one or two years.

increase popularity of cryptocurrencies is the potential to change incentives in shared computing systems. With cryptocurrencies, any spare computation can be used for mining. This has already become a problem, with some websites and apps mining cryptocurrency on users' computers and phones [105]. In shared datacenters within a company or lab, users may suffer retaliation for doing this, but in a less restrictive collaborative environment, users have incentives to turn spare capacity into profit. In this scenario, the idea of considering long-term fairness introduced in Chapter 4 may also be explored to help solving this problem.

Bibliography

[1] JAFFE, J., "Bottleneck Flow Control," *IEEE Transactions on Communications*, v. 29, n. 7, pp. 954–962, Jul. 1981.

[2] GHODSI, A., ZAHARIA, M., HINDMAN, B., *et al.*, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types." In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, pp. 323–336, Mar. 2011.

[3] CHOWDHURY, M., ZHONG, Y., STOICA, I., "Efficient Coflow Scheduling with Varys." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '14, pp. 443–454, Aug. 2014.

[4] GRANDL, R., CHOWDHURY, M., AKELLA, A., *et al.*, "Altruistic Scheduling in Multi-Resource Clusters." In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pp. 65–80, Nov. 2016.

[5] PROCACCIA, A. D., "Cake Cutting: Not Just Child's Play," *Communications of the ACM*, v. 56, n. 7, pp. 78–87, Jul. 2013.

[6] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., *et al.*, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, pp. 295–308, Mar. 2011.

[7] JAIN, R. K., CHIU, D.-M. W., HAWE, W. R., "A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems." Technical Report DEC-TR-301, Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA, Sep. 1984.

[8] JOE-WONG, C., SEN, S., LAN, T., *et al.*, "Multiresource Allocation: Fairness-Efficiency Tradeoffs in a Unifying Framework," *IEEE/ACM Transactions on Networking*, v. 21, n. 6, pp. 1785–1798, Dec. 2013.

[9] WANG, W., LI, B., LIANG, B., "Dominant Resource Fairness in Cloud Computing Systems with Heterogeneous Servers." In: *Proceedings of the IEEE Conference on Computer Communications*, INFOCOM 2014, pp. 583–591, Apr. 2014.

[10] DIXIT, A., PRAKASH, P., HU, Y. C., *et al.*, "On the Impact of Packet Spraying in Data Center Networks." In: *Proceedings of the IEEE Conference on Computer Communications*, INFOCOM 2013, pp. 2130–2138, Apr. 2013.

[11] SEKAR, V., EGI, N., RATNASAMY, S., *et al.*, "Design and Implementation of a Consolidated Middlebox Architecture." In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, pp. 323–336, Apr. 2012.

[12] SHERRY, J., HASAN, S., SCOTT, C., *et al.*, "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '12, pp. 13–24, Aug. 2012.

[13] CHIOSI, M., CLARKE, D., PETER WILLIS, A. R., *et al.*, "Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action." Technical report, European Telecommunications Standards Institute, Oct. 2012. Available at: <`https://portal.etsi.org/NFV/NFV_White_Paper.pdf`>.

[14] HWANG, J., RAMAKRISHNAN, K. K., WOOD, T., "NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms." In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, pp. 445–458, Apr. 2014.

[15] KATSIKAS, G. P., BARBETTE, T., KOSTIĆ, D., *et al.*, "Metron: NFV Service Chains at the True Speed of the Underlying Hardware." In: *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pp. 171–186, Apr. 2018.

[16] KULKARNI, S. G., ZHANG, W., HWANG, J., *et al.*, "NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pp. 71–84, Aug. 2017.

[17] PALKAR, S., LAN, C., HAN, S., *et al.*, "E2: A Framework for NFV Applications." In: *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pp. 121–136, Oct. 2015.

[18] SHERRY, J., GAO, P. X., BASU, S., *et al.*, "Rollback-Recovery for Middleboxes." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '15, pp. 227–240, Aug. 2015.

[19] SUN, C., BI, J., ZHENG, Z., *et al.*, "NFP: Enabling Network Function Parallelism in NFV." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pp. 43–56, Aug. 2017.

[20] TOOTOONCHIAN, A., PANDA, A., LAN, C., *et al.*, "ResQ: Enabling SLOs in Network Function Virtualization." In: *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pp. 283–297, Apr. 2018.

[21] BARI, F., CHOWDHURY, S. R., AHMED, R., *et al.*, "Orchestrating Virtualized Network Functions," *IEEE Transactions on Network and Service Management*, v. 13, n. 4, pp. 725–739, Dec. 2016.

[22] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., *et al.*, "OpenNF: Enabling Innovation in Network Function Control." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '14, pp. 163–174, Aug. 2014.

[23] KABLAN, M., ALSUDAIS, A., KELLER, E., *et al.*, "Stateless Network Functions: Breaking the Tight Coupling of State and Processing." In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pp. 97–112, Mar. 2017.

[24] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., *et al.*, "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes." In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pp. 227–240, Apr. 2013.

[25] WOO, S., SHERRY, J., HAN, S., *et al.*, "Elastic Scaling of Stateful Network Functions." In: *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pp. 299–312, Apr. 2018.

[26] ABRAMSON, M., MOSER, W. O. J., "More Birthday Surprises," *The American Mathematical Monthly*, v. 77, n. 8, pp. 856–858, Oct. 1970.

[27] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., *et al.*, "Hedera: Dynamic Flow Scheduling for Data Center Networks." In: *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '10, pp. 19–19, Apr. 2010.

[28] FERRAZ, L. H. G., LAUFER, R., MATTOS, D. M., *et al.*, "A High-Performance Two-Phase Multipath scheme for Data-Center Networks," *Computer Networks*, v. 112, pp. 36–51, Jan. 2017.

[29] ALIZADEH, M., YANG, S., SHARIF, M., *et al.*, "pFabric: Minimal Near-Optimal Datacenter Transport." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '13, pp. 435–446, Aug. 2013.

[30] CAO, J., XIA, R., YANG, P., *et al.*, "Per-Packet Load-Balanced, Low-Latency Routing for Clos-Based Data Center Networks." In: *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pp. 49–60, Dec. 2013.

[31] HANDLEY, M., RAICIU, C., AGACHE, A., *et al.*, "Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pp. 29–42, Aug. 2017.

[32] ZHANG, H., ZHANG, J., BAI, W., *et al.*, "Resilient Datacenter Load Balancing in the Wild." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pp. 253–266, Aug. 2017.

[33] COUTO, R. S., CAMPISTA, M. E. M., COSTA, L. H. M. K., "A Reliability Analysis of Datacenter Topologies." In: *Proceedings of the IEEE Global Communications Conference*, GLOBECOM 2012, pp. 1890–1895, Dec. 2012.

[34] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator." In: *Proceedings of the 4th ACM Symposium on Cloud Computing*, SoCC '13, pp. 5:1–5:16, Oct. 2013.

[35] REISS, C., TUMANOV, A., GANGER, G. R., *et al.*, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis." In: *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, pp. 7:1–7:13, Oct. 2012.

[36] RADUNOVIC, B., LE BOUDEC, J.-Y., "A Unified Framework for Max-Min and Min-Max Fairness With Applications," *IEEE/ACM Transactions on Networking*, v. 15, n. 5, pp. 1073–1083, Oct. 2007.

[37] DOLEV, D., FEITELSON, D. G., HALPERN, J. Y., *et al.*, "No Justified Complaints: On Fair Sharing of Multiple Resources." In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pp. 68–75, Jan. 2012.

[38] BONALD, T., ROBERTS, J., "Multi-Resource Fairness: Objectives, Algorithms and Performance." In: *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, pp. 31–42, Jun. 2015.

[39] BODIK, P., FOX, A., FRANKLIN, M. J., *et al.*, "Characterizing, Modeling, and Generating Workload Spikes for Stateful Services." In: *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pp. 241–252, Jun. 2010.

[40] SADOK, H., CAMPISTA, M. E. M., COSTA, L. H. M. K., "Per-Packet Load Balancing for Multi-Core Middleboxes," (poster) 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI '18, Apr. 2018. Available at: <`https://www.gta.ufrj.br/ftp/gta/TechReports/SCC18a.pdf`>.

[41] SADOK, H., CAMPISTA, M. E. M., COSTA, L. H. M. K., "A Case for Spraying Packets in Software Middleboxes." In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets-XVII, Nov. 2018. (To appear).

[42] SADOK, H., CAMPISTA, M. E. M., COSTA, L. H. M. K., "Stateful Dominant Resource Fairness: Considering the Past in a Multi-Resource Allocation." In: *Proceedings of the 17th International IFIP TC6 Networking Conference*, IFIP Networking 2018, pp. 415–423, May 2018.

[43] CLARK, D. D., "The Design Philosophy of the DARPA Internet Protocols." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '88, pp. 106–114, Aug. 1988.

[44] CARPENTER, B., BRIM, S., "Middleboxes: Taxonomy and Issues." RFC 3234, Internet Engineering Task Force, Feb. 2002. Available at: <`https://www.rfc-editor.org/rfc/rfc3234.txt`>.

[45] SALTZER, J. H., REED, D. P., CLARK, D. D., "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, v. 2, n. 4, pp. 277–288, Nov. 1984.

[46] LANGLEY, A., RIDDOCH, A., WILK, A., *et al.*, "The QUIC Transport Protocol: Design and Internet-Scale Deployment." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pp. 183–196, Aug. 2017.

[47] PANDA, A., HAN, S., JANG, K., *et al.*, "NetBricks: Taking the V out of NFV." In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pp. 203–216, Nov. 2016.

[48] DPDK. "Data Plane Development Kit." 2018. Available at: <`https://dpdk.org`>.

[49] RIZZO, L., "netmap: A Novel Framework for Fast Packet I/O." In: *Proceedings of the 2012 USENIX Annual Technical Conference*, ATC '12, pp. 101–112, Jun. 2012.

[50] NTOP. "PF_RING ZC (Zero Copy)." 2018. Available at: <`https://www.ntop.org/guides/pf_ring/zc.html`>.

[51] GALLENMÜLLER, S., EMMERICH, P., WOHLFART, F., *et al.*, "Comparison of Frameworks for High-Performance Packet IO." In: *Proceedings of the 11th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pp. 29–38, May 2015.

[52] LINUX. "packet - packet interface on device level." 2018. Available at: <`http://man7.org/linux/man-pages/man7/packet.7.html`>.

[53] RUPP, K. "42 Years of Microprocessor Trend Data." Feb. 2018. Available at: <`https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`>. retrieved 08/09/2018.

[54] DREPPER, U., "What Every Programmer Should Know About Memory." Technical report, Ulrich Drepper Home Page, Nov. 2007. Available at: <`https://www.akkadia.org/drepper/cpumemory.pdf`>.

[55] SCHWARZKOPF, M., BAILIS, P., "Research for Practice: Cluster Scheduling for Datacenters," *Communications of the ACM*, v. 61, n. 5, pp. 50–53, Apr. 2018.

[56] CHOUDHURY, D. G., PERRETT, T., "Designing Cluster Schedulers for Internet-Scale Services," *Communications of the ACM*, v. 61, n. 6, pp. 34–40, May 2018.

[57] VERMA, A., PEDROSA, L., KORUPOLU, M., *et al.*, "Large-Scale Cluster Management at Google with Borg." In: *Proceedings of the 10th European Conference on Computer Systems*, EuroSys '15, pp. 18:1–18:17, Apr. 2015.

[58] WIDE PROJECT. "MAWI Working Group Traffic Archive: `samplepoint-F`." May 2018. Available at: <`http://mawi.wide.ad.jp/mawi/`>.

[59] GUO, L., MATTA, I., "The War Between Mice and Elephants." In: *Proceedings of the 9th International Conference on Network Protocols*, ICNP '01, pp. 180–188, Nov. 2001.

[60] HAN, S., JANG, K., PANDA, A., *et al.*, "SoftNIC: A Software NIC to Augment Hardware." Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[61] DIGITAL CORPORA. "Digital Corpora: M57-Patents Scenario." 2018. Available at: <`https://digitalcorpora.org/corpora/scenarios/m57-patents-scenario`>.

[62] JAMSHED, M., MOON, Y., KIM, D., *et al.*, "mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes." In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pp. 113–129, Mar. 2017.

[63] KOHLER, E., MORRIS, R., CHEN, B., *et al.*, "The Click Modular Router," *ACM Transactions on Computer Systems*, v. 18, n. 3, pp. 263–297, Aug. 2000.

[64] DPDK. "IXGBE Driver." 2018. Available at: <`https://doc.dpdk.org/guides/nics/ixgbe.html`>.

[65] INTEL. "Intel 82599 10 GbE Controller Datasheet." 2016.

[66] INTEL. "Intel Ethernet Controller X710/XXV710/XL710 Datasheet." 2018.

[67] WOO, S., JEONG, E., PARK, S., *et al.*, "Comparison of Caching Strategies in Modern Cellular Backhaul Networks." In: *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pp. 319–332, Jun. 2013.

[68] EMMERICH, P., GALLENMÜLLER, S., RAUMER, D., *et al.*, "MoonGen: A Scriptable High-Speed Packet Generator." In: *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pp. 275–287, Oct. 2015.

[69] IPERF3. "iperf3." 2018. Available at: <`https://software.es.net/iperf/`>.

[70] YU, X., FENG, W.-C., YAO, D. D., *et al.*, "O^{3}FA: A Scalable Finite Automata-Based Pattern-Matching Engine for Out-of-Order Deep Packet Inspection." In: *Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems*, ANCS '16, pp. 1–11, Mar. 2016.

[71] GOOGLE. "HTTPS encryption on the web." 2018. Available at: <`https://transparencyreport.google.com/https`>. retrieved 07/02/2018.

[72] LET'S ENCRYPT. "Let's Encrypt Stats." 2018. Available at: <`https://letsencrypt.org/stats/`>. retrieved 07/02/2018.

[73] ARASHLOO, M. T., GHOBADI, M., REXFORD, J., *et al.*, "HotCocoa: Hardware Congestion Control Abstractions." In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pp. 108–114, Nov. 2017.

[74] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., *et al.*, "Azure Accelerated Networking: SmartNICs in the Public Cloud." In: *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '18, pp. 51–66, Apr. 2018.

[75] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., *et al.*, "NetFPGA – An Open Platform for Gigabit-Rate Network Switching and Routing." In: *Proceedings of the IEEE International Conference on Microelectronic Systems Education*, MSE '07, pp. 160–161, Jun. 2007.

[76] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., *et al.*, "CONGA: Distributed Congestion-Aware Load Balancing for Datacenters." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '14, pp. 503–514, Aug. 2014.

[77] HE, K., ROZNER, E., AGARWAL, K., *et al.*, "Presto: Edge-Based Load Balancing for Fast Datacenter Networks." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '15, pp. 465–478, Aug. 2015.

[78] KANDULA, S., KATABI, D., SINHA, S., *et al.*, "Dynamic Load Balancing Without Packet Reordering," *SIGCOMM Computer Communication Review*, v. 37, n. 2, pp. 51–62, Mar. 2007.

[79] MITZENMACHER, M., "The Power of Two Choices in Randomized Load Balancing," *IEEE Transactions on Parallel and Distributed Systems*, v. 12, n. 10, pp. 1094–1104, Oct. 2001.

[80] ZHANG, Y., ANWER, B., GOPALAKRISHNAN, V., *et al.*, "ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining." In: *Proceedings of the Symposium on SDN Research*, SOSR '17, pp. 143–149, Apr. 2017.

[81] PARKES, D. C., PROCACCIA, A. D., SHAH, N., "Beyond Dominant Resource Fairness," *ACM Transactions on Economics and Computation*, v. 3, n. 1, pp. 3:1–3:22, Mar. 2015.

[82] KASH, I., PROCACCIA, A. D., SHAH, N., "No Agent Left Behind: Dynamic Fair Division of Multiple Resources," *Journal of Artificial Intelligence Research*, v. 51, n. 1, pp. 579–603, Sep. 2014.

[83] GHODSI, A., SEKAR, V., ZAHARIA, M., *et al.*, "Multi-Resource Fair Queueing for Packet Processing." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '12, pp. 1–12, Aug. 2012.

[84] OPPENHEIM, A. V., SCHAFER, R. W., BUCK, J. R., 1999, *Discrete-Time Signal Processing*. 2nd ed. , Prentice-Hall.

[85] BAYER, R., "Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms," *Acta Informatica*, v. 1, n. 4, pp. 290–306, Dec. 1972.

[86] GUIBAS, L. J., SEDGEWICK, R., "A Dichromatic Framework for Balanced Trees." In: *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS 1978, pp. 8–21, Oct. 1978.

[87] GUTMAN, A., NISAN, N., "Fair Allocation Without Trade." In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems – Volume 2*, AAMAS '12, pp. 719–728, Jun. 2012.

[88] FRIEDMAN, E., GHODSI, A., PSOMAS, C.-A., "Strategyproof Allocation of Discrete Jobs on Multiple Machines." In: *Proceedings of the 15th ACM Conference on Economics and Computation*, EC '14, pp. 529–546, Jun. 2014.

[89] ZARCHY, D., HAY, D., SCHAPIRA, M., "Capturing Resource Tradeoffs in Fair Multi-Resource Allocation." In: *Proceedings of the IEEE Conference on Computer Communications*, INFOCOM 2015, pp. 1062–1070, Apr. 2015.

[90] BONALD, T., ROBERTS, J., VITALE, C., "Convergence to Multi-Resource Fairness Under End-to-End Window Control." In: *Proceedings of the IEEE Conference on Computer Communications*, INFOCOM 2017, May 2017.

[91] FRIEDMAN, E., PSOMAS, C.-A., VARDI, S., "Controlled Dynamic Fair Division." In: *Proceedings of the 2017 ACM Conference on Economics and Computation*, EC '17, pp. 461–478, Jun. 2017.

[92] WANG, W., LIANG, B., LI, B., "Low Complexity Multi-Resource Fair Queueing with Bounded Delay." In: *Proceedings of the IEEE Conference on Computer Communications*, INFOCOM 2014, pp. 1914–1922, Apr. 2014.

[93] CHEN, C., WANG, W., ZHANG, S., *et al.*, "Cluster Fair Queueing: Speeding Up Data-Parallel Jobs with Delay Guarantees." In: *Proceedings of the IEEE Conference on Computer Communications*, INFOCOM 2017, May 2017.

[94] BASCH, J., *Kinetic Data Structures*. Ph.D. dissertation, Stanford University, Stanford, CA, USA, Jun. 1999.

[95] DA FONSECA, G. D., DE FIGUEIREDO, C. M. H., CARVALHO, P. C. P., "Kinetic Hanger," *Information Processing Letters*, v. 89, n. 3, pp. 151–157, Feb. 2004.

[96] COUTO, R. S., SADOK, H., CRUZ, P., *et al.*, "Building an IaaS cloud with droplets: a collaborative experience with OpenStack," *Journal of Network and Computer Applications*, v. 117, pp. 59–71, Sep. 2018.

[97] HENNESSY, J. L., PATTERSON, D. A., 2017, *Computer Architecture: A Quantitative Approach*. 6th ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc.

[98] JOUPPI, N. P., YOUNG, C., PATIL, N., *et al.*, "A Domain-specific Architecture for Deep Neural Networks," *Communications of the ACM*, v. 61, n. 9, pp. 50–59, Aug. 2018.

[99] BOSSHART, P., DALY, D., GIBB, G., *et al.*, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Computer Communication Review*, v. 44, n. 3, pp. 87–95, Jul. 2014.

[100] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., *et al.*, "Programmable Packet Scheduling at Line Rate." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '16, pp. 44–57, Aug. 2016.

[101] BAREFOOT NETWORKS, "The World's Fastest & Most Programmable Networks." Technical report, Barefoot Networks, 2016. Available at: <https://www.barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/>.

[102] LIU, J., HALLAHAN, W., SCHLESINGER, C., *et al.*, "p4v: Practical Verification for Programmable Data Planes." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pp. 490–503, Aug. 2018.

[103] VAQUERO, L. M., RODERO-MERINO, L., "Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing," *SIGCOMM Computer Communication Review*, v. 44, n. 5, pp. 27–32, Oct. 2014.

[104] CRUZ, P., PACHECO, R. G., COUTO, R. S., *et al.*, "SensingBus: Using Bus Lines and Fog Computing for Smart Sensing the City," *IEEE Cloud Computing*, 2018.

[105] GOODIN, D. "A surge of sites and apps are exhausting your CPU to mine cryptocurrency." Oct. 2017. Available at: <https://arstechnica.com/information-technology/2017/10/a-surge-of-sites-and-apps-are-exhausting-your-cpu-to-mine-cryptocurrency/>.