Network-Layer Protocols for Data Center Scalability

Yoann Desmouceaux

université PARIS-SACLAY

NNT : 2019SACLX011

Thèse de doctorat

Network-Layer Protocols for Data Center Scalability

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'École Polytechnique

Ecole doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Réseaux, information et communications

Thèse présentée et soutenue à Palaiseau, le 10 avril 2019, par

YOANN DESMOUCEAUX

Composition du Jury :

Marceau Coupechoux Professeur, Télécom ParisTech	Président
Walid Dabbous Directeur de Recherche, INRIA Sophia Antipolis	Rapporteur
Olivier Bonaventure Professeur, Université Catholique de Louvain	Rapporteur
Pascale Minet Chargée de Recherche, INRIA Paris	Examinatrice
Sonia Vanier Maître de Conférences, Université Paris-1 Panthéon Sorbonne	Examinatrice
Thomas Heide Clausen Professeur, École Polytechnique	Directeur de thèse
Mark Townsley Fellow, Cisco Systems; Professeur Chargé de Cours, École Polytechnique	Co-encadrant

Network-Layer Protocols for Data Center Scalability

Yoann Desmouceaux

2019

 This work was conducted while the author was employed by Cisco Systems France. It benefited from support from French CIFRE grant #2015/1293.

Abstract

With the development of demand for computing resources, data center architectures are growing both in scale and in complexity. In this context, this thesis takes a step back as compared to traditional network approaches, and shows that providing generic primitives directly within the network layer is a great way to improve efficiency of resource usage, and decrease network traffic and management overhead. Using two advanced network architectures, Segment Routing (SR) and Bit-Indexed Explicit Replication (BIER), network layer protocols are designed and analyzed to provide three high-level functions: (1) task mobility, (2) reliable content distribution and (3) load-balancing.

First, task mobility is achieved by using SR to provide a zero-loss virtual machine migration service. This then opens the opportunity for studying how to orchestrate task placement and migration while aiming at (i) maximizing the inter-task throughput, while (ii) maximizing the number of newly-placed tasks, but (iii) minimizing the number of tasks to be migrated. Second, reliable content distribution is achieved by using BIER to provide a reliable multicast protocol, in which retransmissions of lost packets are targeted towards the precise set of destinations having missed that packet, thus incurring a minimal traffic overhead. To decrease the load on the source link, this is then extended to enable retransmissions by local peers from the same group, with SR as a helper to find a suitable retransmission candidate. Third, load-balancing is achieved by way of using SR to distribute queries through several application candidates, each of which taking local decisions as to whether to accept those, thus achieving better fairness as compared to centralized approaches. The feasibility of hardware implementation of this approach is investigated, and a solution using covert channels to transparently convey information to the load-balancer is implemented for a state-of-the-art programmable network card. Finally, the possibility of providing autoscaling as a network service is investigated: by letting queries go through a fixed chain of applications using SR, autoscaling is triggered by the last instance, depending on its local state.

Keywords — Data-center networking, Task Mobility, Multicast, Load Balancing, Segment Routing

Résumé

Du fait de la croissance de la demande en ressources de calcul, les architectures de centres de données gagnent en taille et complexité. Dès lors, cette thèse prend du recul par rapport aux architectures réseaux traditionnelles, et montre que fournir des primitives génériques directement à la couche réseau permet d'améliorer l'utilisation des ressources, et de diminuer le trafic réseau et le surcoût administratif. Deux architectures réseaux avancées, Segment Routing (SR) et Bit-Indexed Explicit Replication (BIER), sont utilisées pour construire et analyser des protocoles de couche réseau, afin de fournir trois primitives : (1) mobilité des tâches, (2) distribution fiable de contenu, et (3) équilibre de charge.

Premièrement, pour la mobilité des tâches, SR est utilisé pour fournir un service de migration de machine virtuelles sans perte. Cela ouvre l'opportunité d'étudier comment orchestrer le placement et la migration de tâches afin de (i) maximiser le débit inter-tâches, tout en (ii) maximisant le nombre de nouvelles tâches placées, mais (iii) minimisant le nombre de tâches migrées. Deuxièmement, pour la distribution fiable de contenu, BIER est utilisé pour fournir un protocole de multicast fiable, dans lequel les retransmissions de paquets perdus sont cibleés vers l'ensemble précis de destinations n'ayant pas reçu ce paquet : ainsi, le surcoût de trafic est minimisé. Pour diminuer la charge sur la source, cette approche est étendue en rendant possibles des retransmissions par des pairs locaux, utilisant SR pour trouver un pair capable de retransmettre. Troisièmement, pour l'équilibre de charge, SR est utilisé pour distribuer des requêtes à travers plusieurs applications candidates, chacune prenant une décision locale pour accepter ou non ces requêtes, fournissant ainsi une meilleure équité de répartition comparé aux approches centralisées. La faisabilité d'une implémentation matérielle de cette approche est étudiée, et une solution (utilisant des canaux cachés pour transporter de façon invisible de l'information vers l'équilibreur) est implémentée pour une carte réseau programmable de dernière génération. Finalement, la possibilité de fournir de l'équilibrage automatique comme service réseau est étudiée : en faisant passer (avec SR) des requêtes à travers une chaîne fixée d'applications, l'équilibrage est initié par la dernière instance, selon son état local.

Mots-clefs — Réseaux de centres de données, Mobilité, Multicast, Équilibre de charge, Segment Routing

Remerciements

Tout d'abord, je tiens à remercier chaudement Thomas Clausen pour avoir été mon directeur de thèse durant ces trois années. Thomas a été bien plus qu'un directeur de thèse : dépassant largement les attentes que l'on peut avoir d'un superviseur, il a su être toujours disponible, tant pour fournir de l'aide et du soutien, des discussions scientifiques (et non scientifiques) enrichissantes et des conseils avisés, que des opportunités de collaboration, de l'humour, mais aussi beaucoup d'encre rouge. Je n'aurais pas pu aller au bout de cette thèse sans son aide précieuse. Mes remerciements vont ensuite naturellement à Mark Townsley, qui m'a permis d'effectuer cette thèse au sein de Cisco et sans lequel la dimension industrielle de cette thèse n'aurait pas pu exister, et qui m'a supervisé tout au long de ces trois ans. Travailler avec Mark fut un plaisir, du fait de sa sympathie et de son enthousiasme naturels, de sa capacité à brasser les idées au cours de séances de *whiteboarding* mémorables, tout en étant capable d'en extraire ensuite les idées essentielles tant industriellement qu'académiquement.

Je remercie mes rapporteurs, Olivier Bonaventure et Walid Dabbous, pour la lecture attentive qu'ils ont faite de mon manuscrit, pour avoir pris le temps d'écrire des rapports très détaillés et pertinents, et pour avoir pu se déplacer de loin pour pouvoir prendre part à ma soutenance. Merci également aux autres membres du jury, Marceau Coupechoux, Pascale Minet et Sonia Vanier, pour leur présence à ma soutenance, leurs enthousiasme et leurs questions pointues.

Je tiens ensuite à remercier mes collègues de Cisco, notamment les autres (pour certains, ex-) thésards, pour ces années passées ensemble : Marcel Enguehard et son obsession pour les filtres LRU, Mohammed "Momo" Hawari et sa fascination pour les instructions `x86_64` étranges, Guillaume "Gruty" Ruty et les 100 gamelles, Jacques Samain et ses farces perpétuelles. L'ambiance au bureau n'aurait pas été la même sans les autres "jeunes" : Aloÿs Augustin, Mohsin Kazmi, Victor Nguyen, Pierre Pfister, Wenqin Shao, Nathan Skrzypczak, que je remercie pour leur bonne humeur et leur participation au concours de babyfoot. Je porte une mention spéciale à Jérôme Tollet pour son rire inimitable et son humour intarissable. Merci à tous les autres avec qui j'ai collaboré ou tout simplement discuté, en espérant ne pas en oublier : Jordan Augé, Giovanna Carofiglio, Enzo Fenoglio, Alain Fiocco, Benoît Ganne, Guillaume Ladhuie, Malycia Ly, Luca Muscariello, Guillaume Sauvage de Saint Marc, Hassen Siad, André Surcouf, Axel Taldir, Éric Vyncke. Ces trois années ont aussi vu se succéder plusieurs générations de stagiaires, qui ont apporté de la fraîcheur chaque année : Zacharie Brodard, Basile Bruneau, Clément Fischer ; Clément Durand, Pierre-Jean Grenier, Maxime Larcher, Samuel Perez ; Thomas Feltin, Benoît Pit--Claudel, Andi Sai, Arthur Toussaint, Zhiyuan Yao ; Clemens Kunst, Alexandre Poirrier, Guillaume Solignac. Enfin, tout ceci n'aurait pas pu avoir lieu sans l'aide inestimable de Carole Reynaud, qui a su déjouer les situations administratives les plus retorses sans perdre de sa bonne humeur.

Je remercie également mes collègues du LIX, Juan-Antonio Cordero-Fuertes et Jiazi Yi, qui ont rendu le temps passé sur le plateau (ou dans le RER B !) bien plus agréable. Merci aux jeunes qui sont passés par le labo pour leur bonne humeur et leur enthousiasme, notamment Hassan Hankir et Benedikt Tegethoff. Merci aussi à Bernadette Charron-Bost et à Erwan Le Pennec pour des discussions très intéressantes. Enfin, merci à Évelyne Rayssac, qui m'a évité bien des tracas administratifs avec l'X.

Je suis évidemment redevable envers l'ensemble de mes co-auteurs pour leur travail rigoureux et le temps passé à discuter et relire les brouillons d'articles, et sans lesquels les publications mentionnées dans cette thèse n'auraient pas vu le jour. La plupart ont déjà été cités plus haut, mais je les liste à nouveau ici : Thomas Clausen, Juan-Antonio Cordero-Fuertes, Marcel Enguehard, Victor Nguyen, Pierre Pfister, Benoît Pit--Claudel, Wenqin Shao, Jérôme Tollet, Sonia Toubaline, Mark Townsley, Éric Vyncke.

Pour finir, je remercie ma famille pour leur présence tout au long de cette épopée que furent ces études (prépa, école, master, thèse...). Merci particulièrement à ma mère de m'avoir encouragé à faire une thèse malgré mon indécision initiale. Merci encore à ceux de ma famille et de mes amis qui ont pu venir jusqu'à Palaiseau pour ma soutenance. Je suis aussi très reconnaissant envers Nouchka de s'être tenue relativement tranquille lors de la nuit qui précéda ma soutenance.

Merci à Charlotte d'avoir été là du début à la fin, d'avoir supporté mon verbiage sur un domaine qui t'est éloigné, de m'avoir soutenu quand ça n'allait pas, et d'avoir partagé de beaux moments quand ça allait.

Contents

Part I

Introduction

Chapter 1

Introduction

The development of virtualization, where computers are emulated and/or sharing an isolated portion of the hardware by way of *Virtual Machines* (VMs) [1], or run as isolated entities (*containers* [2]) within the same operating system kernel, has accelerated the development of *microservices* [3] and led to the commoditization of compute resources[1]. Thus, data centers are becoming "miniature" Internets, running on top of specialized network architectures, and possibly relying on overlay technologies to provide transparent services to the hosted applications [4]. Given this, architecting network paradigms suited for data center environments is a complex challenge, both from (i) an operational [5] and (ii) an energy consumption perspective [6] – due to (i) the complexity of orchestration and the richness of available architectures, and (ii) the need to minimize power consumption while maintaining quality of service.

In this context, the question explored in this thesis is whether, and how, it is possible to provide certain generic data-center primitives (task migration, reliable content distribution, load balancing, auto-scaling) as network services, thereby allowing for optimizing resources while incurring low operational complexity. This is accomplished by studying how certain standardized network frameworks [7, 8] can be extended to express those primitives directly at the network layer, while operating transparently to the applications.

The remainder of this introductory chapter is structured as follows. Section 1.1 introduces and reviews traditional data center architectures, from network protocols through topologies, and to management frameworks. Then, section 1.2 introduces two network paradigms, Segment Routing [7] and Bit-Indexed Explicit Replication [8], which enrich network layers by allowing sources to include further expressiveness in emitted packets, than solely source and destinations. Finally, section 1.3 argues for the usage of these new paradigms to augment data center architectures, by providing network protocols that are transparent to the applications, while naturally providing (i) task mobility, (ii) reliable content distribution, and (iii) load-balancing and autoscaling. The chapter is concluded by section 1.4, which provides generic assumptions, definitions and notations that will be used throughout this thesis.

1.1 Background

This section introduces background on data centers: section 1.1.1 introduces traditional network protocols used in the Internet and in data centers, and section 1.1.2 reviews data center topologies and architectures.

1.1.1 Network Protocols

Network protocols underlying the Internet are traditionally presented as belonging to several *layers*. Each of these layers (layer n) provides service to the layer above (layer $n+1$) and uses service from the layer below (layer $n-1$), and can transmit messages to and receive messages from those adjacent layers, making network processing effectively separated between independent

[1] According to a 2018 report (`https://www.gartner.com/newsroom/id/3871416`), the public cloud market revenue grew from $154 billion in 2017 to $186 billion in 2018. According to a 2018 survey of 997 IT professionals (`https://www.rightscale.com/lp/2018-state-of-the-cloud-report`), 96% of respondents use private or public clouds.

submodules. For instance, the "Open Systems Interconnection (OSI) model" [9] introduces seven layers – specifically the (i) physical, (ii) data link, (iii) network, (iv) transport, (v) session, (vi) presentation and (vii) application layers. More simply, the usual Internet model comprises four layers [10], and this will be the model used throughout this manuscript:

1. The data-link layer (henceforth, **Layer-2**), whose purpose is both to (i) modulate to and demodulate messages from a physical medium (wired or wireless), and to (ii) control access to that medium, notably by taking care of modulation selection, transmission timing, collisions, retransmissions, and possibly authentication and encryption. Nodes on a single data-link are able to communicate with each other, be they or not connected to an outer network.

2. The network layer (henceforth, **Layer-3**), whose purpose is to carry messages across multiple networks inter-connected by way of routers.

3. The transport layer (henceforth, **Layer-4**), whose purpose is to create logical channels between two distant hosts and transmit messages across those, possibly handling acknowledgement of data transmission, retransmissions, reordering, flow control and congestion control.

4. The application layer (henceforth, **Layer-7**), which allows one-to-one or one-to-many applications running on different endpoints to communicate according to an application-tailored protocol, by transparently using the layers below.

Several protocols exist to provide the functions of each of these layers. As a typical protocol stack for traditional (and, to a certain extent, legacy) data center networks, Ethernet [11] is used as Layer-2 protocol, typically to connect physical machines belonging to a single rack[2]. On top of this, the Internet Protocol (IP) [13,14] is used as Layer-3 protocol, enabling end-to-end connections of hosts within the data center, and allowing interconnection to the Internet. In addition, routing protocols are usually run to build the tables of the routers connecting the different nodes of the data centers. For instance, the Border Gateway Protocol (BGP) [15], initially designed to enable routing between the different Autonomous Systems comprising the Internet, is one the possible routing protocols for data center environments [16]. Then, the Transmission Control Protocol (TCP) [17] is usually used as Layer-4 protocol, allowing for reliable, ordered transmission of streams of data. Finally, one example of a Layer-7 protocol running on top of the data-center stack is the HyperText Transfer Protocol (HTTP) [18] (or its secure version, HTTPS [19]). Initially designed for transfer of files for web browsing, the emergence of REpresentational State Transfer (REST) APIs [20] makes HTTP also used as a data manipulation and retrieval protocol.

Although the above-mentioned protocol stack has been widely used across the Internet and in data center premises, it was designed for an "idealized" network, with the *end-to-end principle* [21] in mind. According to this principle, network functions (reachability, encryption, reliability, *etc.*) should, when possible, be fulfilled by the communicating endpoints of a logical transmission. In such an idealized network, each host is equally reachable and can equally perform "client" or "server" functions. However, this principle may not always be true, for multiple reasons. First, IPv4 address exhaustion[3] makes the use of private addressing schemes and of Network Address Translation (NAT) devices to translate addresses at the ingress and egress of these private networks prevalent. Second, the progressive shift to a consumer-producer paradigm [22], where end users often want to retrieve content, and data center facilities are often used to serve content, imposes an asymmetry between two types of hosts. Third, data centers (and especially public clouds) are often configured to run multi-tenancy scenarios [23], wherein multiple virtual sub-facilities – each intended to belong to one *tenant*[4] – are sharing the same physical resources (physical machines and physical network).

For these reasons, data center networks usually rely on abstractions to provide multiple virtual flat networks, hiding the complexity of their physical network architecture to provide a consistent and isolated network to each of the multiple tenants [4]. Each tenant runs a set of VMs and is, for example, presented with a virtual Layer-2 network so that, in a way, the end-to-end principle is restored within this network. According to [4], the simplest way to implement these virtual

[2] Although only wired data centers will be studied in this thesis, wireless data center architectures are an active research area [12].

[3] `https://www.icann.org/en/system/files/press-materials/release-03feb11-en.pdf`

[4] For the purpose of this section, a tenant is defined as any entity using some of the resources of a data center for a single purpose, be it an individual, a company, or a logical entity such as a project to which resources are assigned, *etc.*

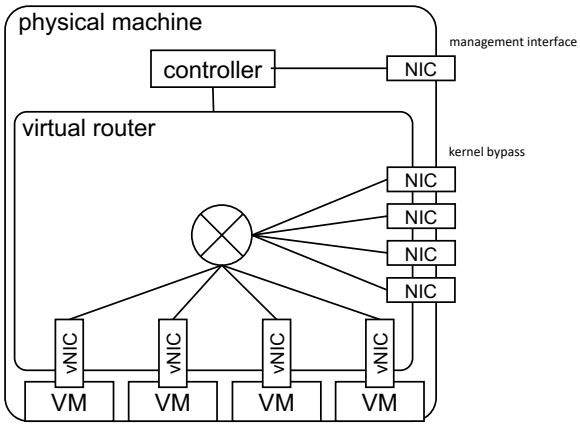


Figure 1.1 – Virtual router operation

networks in data center environments is by using Virtual Local Area Networks (VLANs) [24]: each Ethernet packet carries a 12-bit VLAN identifier, and switches forward packets only between those physical ports which are configured to belong to the same VLAN. However, due to the restricted corresponding identifier space (only allowing for $2^{12} = 4096$ tenants), and the necessity for tenant networks to be included in a single physical Layer-2 network, other protocols have been developed that allow for more flexibility – at the cost of operational complexity. Among those, Virtual eXtensible Local Area Networks (VXLANs) [25] allow for virtual Layer-2 network to span arbitrary topologies, by encapsulating Layer-2 frames within a UDP/IP packet containing a special header enabling a network to be divided among up to $2^{24} \approx 16.8 \cdot 10^6$ tenants. With VXLAN encapsulation, packets can be transmitted within one data center (or across several data centers) by using the underlying routing infrastructure. Another protocol which can be used for data center multi-tenancy is Network Virtualization using Generic Routing Encapsulation (NVGRE) [26], which encapsulates Layer-2 frames within GRE packets, and also allows for up to 2^{24} tenants. Finally, inter-data-center multi-tenancy can be achieved by way of Virtual Private Network (VPN) protocols such as Virtual Private LAN Service (VPLS) [27] or Ethernet VPN (EVPN) [28]. With these protocols, Layer-2 frames belonging to a single tenant are transmitted across a Multi-Protocol Label Switching (MPLS) [29] capable network shared between two data center premises, using MPLS labels to distinguish between tenants.

Along with the virtualization of compute resources, network resources are also often virtualized in order to provide the encapsulation services mentioned above. Virtual routers, such as Open VSwitch (OVS) [30] or the Vector Packet Processor (VPP) [31], are usually run as services in each physical machine, with virtual interfaces connected to each of the containers and/or VMs (possibly belonging to several tenants) hosted in that machine, and with physical interfaces connected to the physical infrastructure of the data center, as depicted in figure 1.1. These virtual routers then properly encapsulate or decapsulate traffic from and to the instances of the tenants, often using kernel-bypass techniques (*e.g.,* DPDK [32]) to provide high-performance packet processing. Other than connecting instances, such virtual routers can also be deployed (usually as virtual machines themselves) for more specific purposes, *e.g.,* firewalling, intrusion detection, address translation, encryption, load-balancing, *etc.*, thus providing Network Function Virtualization (NFV) [33].

As has been described above, these layers of virtualization and encapsulation allow to hide the complexity of the network infrastructure to tenants so that, within each sub-network, the end-to-end principle can apply. However, this comes at the cost of operational complexity, as virtual routers must be deployed, and as the corresponding network configuration (identifier assignments, *etc.*) must be computed and distributed.

1.1.2 Data Center Architectures

According to [5, 6, 35], one of the first data center architectures to have been used is the *three-tiered* architecture described in [34] (figure 1.2). With this architecture, which is essentially a (potentially multi-rooted) tree, machines are grouped into racks connected to a *Top-of-Rack* (ToR)

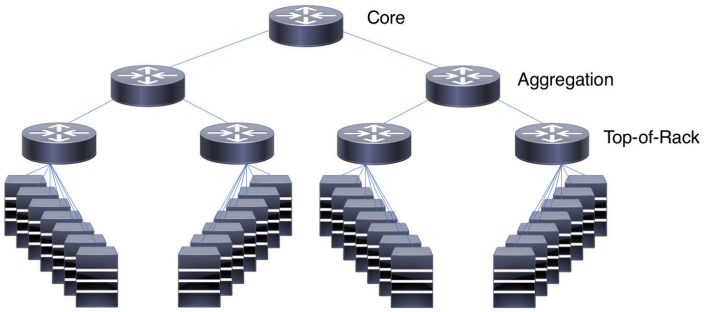


Figure 1.2 – Three-tiered data center network topology [34]

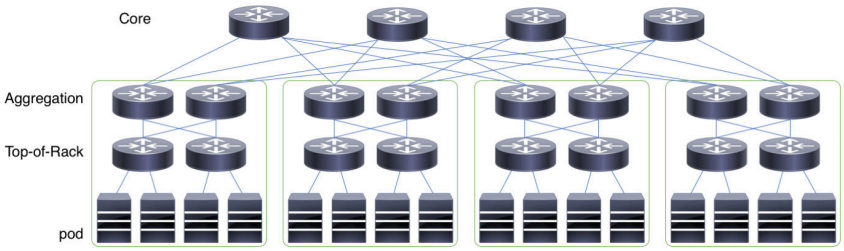


Figure 1.3 – Fat-tree data center network topology [35] with $k = 4$

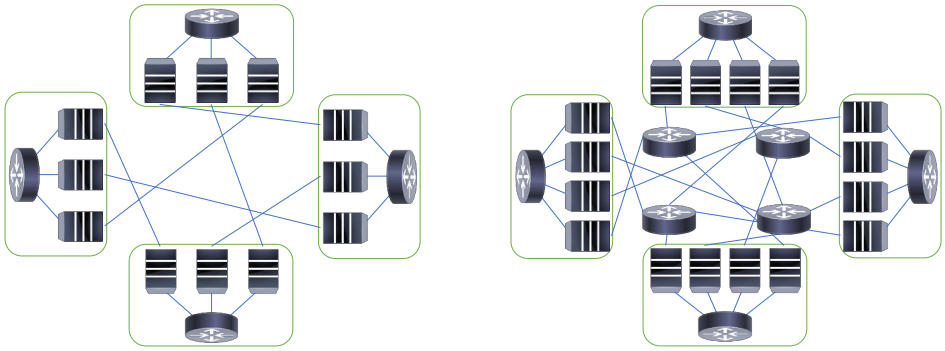
router[5]. Each of these ToR routers is itself connected to one among several *aggregation* routers – thus each aggregation router is the root of a subtree comprising several racks. Finally, aggregation routers are meshed to one or several *core* router(s), usually through links of higher capacity, thus allowing a host in any rack to reach any other host by crossing 1, 3 or 5 routers. These core routers can be connected to an egress router, which acts at the gateway of the data center. A drawback of this topology, as identified in [35], is that the links between core and aggregation routers act as bottlenecks, thus limiting the throughput achievable between two arbitrary hosts when many hosts communicate with other non-local hosts. Overcoming this limitations require increasing the number of high-throughput routers, and therefore incurs an often prohibitive hardware cost.

To address these drawbacks, an important amount of work has consisted of deriving topologies for data centers that are efficient in terms of throughput while minimizing hardware cost and environmental impact [5,6]. In [35], a data center architecture is presented, which aims at reducing the hardware cost of a throughput-efficient[6] network, compared to three-tiered data centers. This is achieved by using a *fat-tree* topology [38] (figure 1.3): with k being an even integer, k core routers are each connected to one router in each of k pods. Each of the k pods comprises two layers of $k/2$ routers: the upper (aggregation) layer is connected with the core layer, and the lower (ToR) layer is connected to hosts. Within a pod, the aggregation layer is meshed with the ToR layer. Finally, each ToR router is connected to $k/2$ hosts. This allows connecting $k^3/4$ hosts with $k(k+1)$ routers while achieving maximum bissection bandwidth. However, the proposed architecture comes at an operational cost, as it imposes a custom addressing scheme, along with a hardware modification of the routing lookup mechanism to allow for prefix- and suffix-based routing. Furthermore, it requires a custom centralized flow scheduler in order to achieve full throughput.

VL2 [39] is another proposed hierarchical data center architecture, which uses a Clos topology [40] and provides separation of identifiers and locators through two separate address spaces and IP-in-IP encapsulation, while presenting a Layer-2 semantic to the whole overlay network. A centralized directory maintains a mapping between applications identifiers and locators of their ToR routers, and a shim layer running in each machine provides encapsulation services, while decapsulation is performed at the ToR routers. Source routing is used to balance traffic across

[5]In this section, the word "router" is used to designate devices that can perform Layer-2 and/or Layer-3 forwarding, indifferently.

[6]This is assessed by computing the *bissection bandwidth*. The *bissection width* of a network is defined as the minimal width (number) of links that need to be cut so as to separate the network into two equal parts. The bissection bandwidth then corresponds to the total throughput that these cut links can sustain.



(a) DCell [36] with $n = 3$ and $\ell = 1$ **(b)** BCube [37] with $n = 4$ and $\ell = 1$

Figure 1.4 – Advanced data center network toplogies

core routers randomly. With this architecture, routing tables within routers are only bound to the physical topology (making them lightweight and rarely changing), while applications can be freely dynamically allocated on top of this underlay architecture.

Although the architectures in [35, 39] possess good properties in terms of cost and achievable throughput, they remain hierarchically organized in three layers, which can pose scalability issues for very large data centers – for instance with [35], core routers need to have $\mathcal{O}(n^{1/3})$ ports to build an n-machine data center. To address these scalability issues, recursive architectures have been introduced. Among those, DCell [36] is an architecture using routers with a fixed number of ports. Starting from a small building block of level 0 (a rack of n machines connected to an n-ports router), blocks of level $\ell+1$ are built by connecting $s_\ell + 1$ blocks of level ℓ, where s_ℓ is the number of machines in a block of level ℓ, as depicted in figure 1.4a. This is achieved by realizing a full mesh between sub-blocks: two sub-blocks are connected by picking one machine into each of them, and connecting them to each other. Therefore, machines become parts of the routing infrastructure, with routers only providing rack connectivity. Thus, DCell uses a custom Layer-3 protocol, with a custom forwarding module in the physical machines: this has an operational cost and incurs additional resource (CPU) consumption. Another recursive topology is BCube [37], whose main difference to [36] is that it uses routers not only at the rack level, and provides multiple parallel paths between pairs of machines. With a building block (of level 0) consisting of n machines connected to an n-ports router, a block of level $\ell + 1$ is built by connecting n blocks of level ℓ together, using $n^{\ell+1}$ routers, as depicted in figure 1.4b. Similarly to [36], machines are part of the forwarding infrastructure, and a custom source routing protocol is used to select a server-path between two nodes, with a custom module deployed in each machine.

Another research aspect related to data center architectures is that of traffic management. Indeed, it has been shown that data center traffic is bursty [41] – for instance, one of the data center studied in [41] exhibits a median flow inter-arrival time lower than 250 μs. Given this, architectures have been designed to properly balance traffic among data center links while handling asymmetry and reacting to congestion. For example, Hedera [42] is a centralized architecture which aims at properly balance flows in order to provide near-optimal bissection bandwidth. It consists of a centralized scheduler, which periodically probes ToR routers and extracts characteristics of the larger flows. From this, it estimates the throughput demand of each flow (*i.e.,* the throughput that the flow would be achieving if it was only limited by the interfaces of the sender and receiver), and uses simulate annealing to compute an almost-optimal placement of flows to satisfy these demands. Finally, corresponding rules are pushed to the routers using a Software Defined Network (SDN) controller [43]. Similarly, inter-data-center centralized traffic management architectures have been proposed [44, 45]. Another data-center traffic management framework is Conga [46], which acts at the "flowlet" level rather than making per-flow decisions. Flowlets are defined as bursts of packets sufficiently spaced in time so that they can be sent over different paths without causing packet reodering. With Conga, each intermediate router marks congestion information within a special VXLAN header, before reflecting this to the ToR router from which traffic originates. Each ToR router maintains a per-ToR-router congestion table, and makes traffic placement decisions accordingly. While providing reactive traffic balancing and global congestion knowledge, this architecture requires hardware modifications in the routers (to be able to mark congestion).

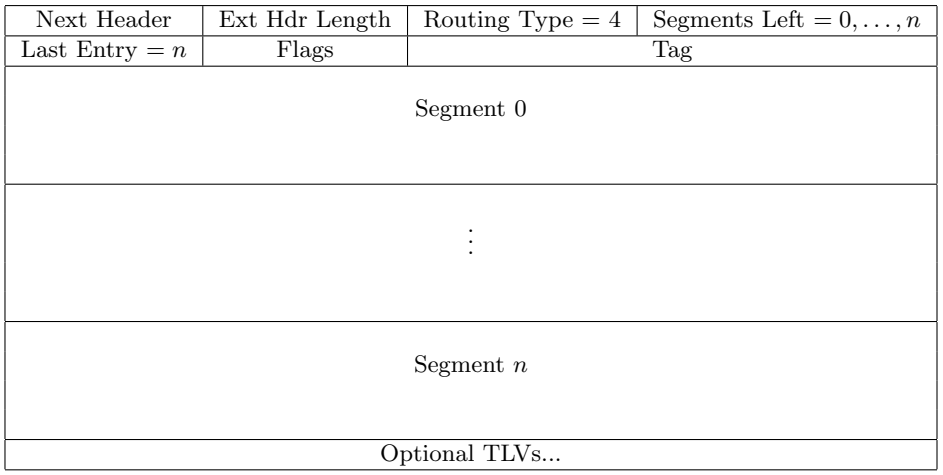


Figure 1.5 – IPv6 Segment Routing Header [47]

Algorithm 1 SR Header Processing at a given Segment Endpoint

```
p ← packet
S ← address of this segment
if p.dst = S and p.nextHeader = 43 and p.routingType = 4 then
    if p.segmentsLeft > 0 then
        apply the function of this segment to p
        p.segmentsLeft ← p.segmentsLeft − 1
        p.dst ← p.segmentsList[p.segmentsLeft]
        p.ttl ← p.ttl − 1
        transmit p over interface for p.dst
    else
        drop the packet
    end if
else  ▷ non-SR packet or packet destined to another router
    process packet normally
end if
```

As described in sections 1.1.1 and 1.1.2, data center architectures have evolved to address scalability issues in terms of topology and traffic management. On the one hand, these evolutions often rely on complex overlays, specific routing schemes, and/or custom network protocol stacks. On the other hand, some protocols have been proposed in order to simplify network operation, by encoding instructions directly in the packets rather than provisioning ad-hoc rules in advance. Section 1.2 describes two such network architectures, and in section 1.3 will describe if and how they can be useful to solve data-center-specific issues.

1.2 Extending the Network Layer

This section introduces two network paradigms, Segment Routing (section 1.2.1) and Bit-Indexed Explicit Replication (section 1.2.2), which will be used throughout this thesis as foundational blocks to provide augmented data center services. A common point to these two architectures is that they rely on source-provided instructions embedded in a given network packet to influence the behavior of further devices (*e.g.,* routers or hosts) that will process and/or forward that packet.

1.2.1 Segment Routing (SR)

Segment Routing (SR) [7] is a network architecture enabling source-routing capabilities in a controlled domain, and standardized by the Internet Engineering Task Force (IETF[7]) [48]. SR is an *architecture*, for which there exists two data-plane *instantiations* or "flavors": (i) an MPLS flavor, where segments are encoded by way of MPLS labels [49], and (ii) an IPv6 flavor [47], where

[7] `https://www.ietf.org`

segments are encoded by way of IPv6 addresses. For the purpose of this manuscript, only the **IPv6** flavour (IPv6 Segment Routing or SRv6) will be considered.

SRv6 allows a given packet to be forwarded along a set of "segments" encoded in the packet header, each of these segments being represented by an IPv6 address, and representing a specific function to be executed on the packet. This is done using an IPv6 extension header, defined in [47] and depicted in figure 1.5. This header carries a list of segments, which the packet should traverse, with the last segment representing the ultimate destination address of the packet. Each segment is divided into a *locator* part (typically, the first 64 bits), and a *function* part (typically, the last 64 bits). The locator is a routable IPv6 prefix, meaning that the packet will be naturally directed to the next segment using the underlying routing infrastructure (possibly taking advantage of ECMP multipath routes), and that the packet can therefore naturally traverse non-SR-capable routers.

An SRv6 header is initialized with a list of segments (in reverse traversal order), with the last segment being the original destination address of the packet. Fields `LastEntry` and `SegmentsLeft` are initialized to the number of segments n, and the destination address of the packet rewritten to that of the segment at position n – *i.e.,* the first segment in traversal order. As described in algorithm 1, upon receiving a packet, a simple SR endpoint will process the packet according to the function instructed in the segment identifier and, if applicable, will initiate forwarding the packet to the next segment. This is done by replacing the destination address of the packet by that of the next segment, and by decreasing the `SegmentsLeft` counter. Of course, if complex behavior is applied by the segment endpoint to the packet (*e.g.,* adding a new header), the simple process described in algorithm 1 needs to be adapted accordingly – for instance, if inserting a new header, lines $6 \rightarrow 8$ of algorithm 1 might not be necessary.

Some example functions that can be instantiated by segments are defined in [50]. The simplest one is the "endpoint" function, denoted `END`, which essentially is a no-op and consists of simply forwarding the packet to the next segment. By simply using a sequence of `END` segments, *source routing* results, that is, a packet follows a source-specified path. Source routing was already defined as a component of IPv6 in its original specification [51] (later superseded by [14]), but was later deprecated for security reasons [52]. Indeed, unrestricted IPv6 source routing could lead to, among others, traffic amplification attacks and network topology exposures [53].

To avoid reproducing these issues, Segment Routing introduces the concept of *SR domain*, *i.e.,* a consistent and autonomous set of nodes controlled by the same administrative entity (*e.g.,* a data center or an ISP network). Within a single domain, it is assumed that all nodes are trusted, in which case packets embedding an SRv6 header can be directly issued and received by hosts. For packets traversing a domain, SR defines the concept of *ingress node*. Upon entering an SR domain by reaching the ingress node, and depending on characteristics of the packets (source address, destination address, *etc.*), a packet will have an SRv6 header applied to it, containing a suitable segment list. The SRv6 header can either be inserted within the packet ("insert mode"), or a new outer IPv6 header can be prepended to the packet ("encap mode"), leaving the original packet intact. Upon reaching the last segment, the SRv6 header can be removed, so that it does not appear in the resulting packet, and so that the packet egresses the domain with its original form. As argued in [47, section 6], this does not have the security drawbacks of the original IPv6 source routing, provided that packets entering the domain and already containing segments pointing to inside the domain are discarded. In cases where SR packets *are* to traverse multiple domains, security can be achieved by the use of a Hash-based Message Authentication Code (HMAC), as described in [47, section 6.3.2].

Beyond the "endpoint" function, other modular functions are defined in [50]. For instances, some functions allow for forwarding the packet according to non-default semantics – `END.X` allows forwarding along a specified interface, and `END.T` forwarding according to a lookup in a specific table. Other functions (`END.DX2`, `END.DX2V`, `END.DT2U`, `END.DX`, `END.DT`) allow decapsulating the SRv6 header and forwarding the resulting according to different lookup strategies (*e.g.,* along a specified interface or according to a lookup in a specified table). Finally, some functions (`END.B6`, `END.BM`) allow insertion of new SR headers to a packet. The variety of these functions indicates how SR can be used for more complex applications than sole source routing.

Notational Note

The SRv6 header format as specified in [47] and depicted in figure 1.5 contains segments in *reverse* traversal order: the first segment encoded in the header is the last traversed, the second encoded is the second-to-last traversed, *etc.* For ease of readability, however, throughout this

manuscript the inverse convention will be adopted. An SR list $(s_1, s_2, \ldots, s_n)$ will thus denote a segment list whose i^{th} segment to be traversed is s_i.

Related Work

SR has been used for different purposes, the first of which is *traffic engineering*. In [54], traffic engineering goals (*e.g.,* "the load on all links should not exceed 90% of their capacity) are considered and expressed in an almost natural (Scala-based) language. An algorithm is introduced to turn these requirements into a set of SR lists instantiating paths between endpoints. In [55], a model of traffic engineering aiming at minimizing link utilization by directing flows through SR lists of length 2 is formulated, and offline and online resolution algorithms are proposed. In [56], SR is used for sub-second traffic engineering. That is, links are continuously monitored, and when congestion (rather than link failure in traditional traffic engineering) is detected, new SR paths are computed to limit link usage. In [57], an incremental deployment of SR within an ISP is considered, and an algorithm to minimize link utilization in a hybrid SR/IP network where the size of the SR domain gradually increases is proposed. Finally, in [58], an approach is proposed to provide traffic engineering within enterprise networks. A custom DNS resolver receives DNS queries augmented with traffic requirements, computes a suitable path and communicates it to the access router, before replying to the client with a corresponding "path ID". The client embeds this path ID as an `END.B6` segment in outgoing packets; upon reaching the access router, this segment triggers insertion of the previously-provisioned path as a new SRv6 header.

Another interesting use case for SR is *service function chaining*. As introduced in [59], it consists of specifying a list of Virtual Network Functions (VNFs) that should be applied to a packet as a set of SR segments. In [60], a Linux kernel module is presented, which allows forwarding SR packets belonging to a service chain, to VMs running SR-unaware VNFs. This is done by decapsulating and re-encapsulating packets, so that VNFs only are exposed to non-SR packets. In [61], an SR-aware VNF is presented, which allows native firewalling of SRv6 packets belonging to a service chain, by way of an extension of the Linux firewall `iptables`. In [62], the SRv6 service chaining abstraction is extended from packets to bytestreams. This is achieved by using a transparent proxy that terminates TCP connections and opens a new TCP stream between each pair of successive segments in the chain.

SR has other use cases which go beyond traffic engineering and service chaining. In [63], SR is used for *network monitoring*, by sending probe packets along a *cycle* of nodes, thus providing a synchronization-free method to detect link failures. Traffic duplication [64] is another use case, where SR is used to send TCP traffic across two parallel paths. When there is spurious packet loss or latency along one of the paths, this helps reducing the flow completion time, while only marginally increasing it when the paths are identical and non-faulty.

Finally, it is noteworthy that in addition to being under standardization at the IETF, SRv6 benefits from an open source implementation in the Linux kernel [65] for both endpoints and routing stacks. This makes SRv6 an open and promising framework for innovative network applications.

1.2.2 Bit-Indexed Explicit Replication (BIER)

Bit-Indexed Explicit Replication (BIER) is a multicast architecture, where packets are sent to a source-specified set of destinations, and standardized by the IETF [8]. Traditional multicast protocols (*e.g.,* Protocol Independent Multicast, PIM [66]) rely on a client-based subscription model: a multicast group address is pre-provisioned, and clients wishing to be part of the group send "join" messages containing that address to their local router. Upon receiving such a message and before forwarding it to the next router leading to the source, routers record state regarding which interfaces lead to a client that has subscribed to the group (therefore collectively building a spanning tree). When receiving packets from the source addressed to the multicast address, routers know on which interface they should be duplicated to reach all destinations. This requirement of per-flow state in the routers as well as the complexity of group management have hindered the deployment of multicast in the Internet [67].

BIER, on the other hand, alleviates this complexity by removing the need for per-flow state in intermediate routers. Rather, it lets multicast packets themselves carry information about the destinations that they need to be carried over to. While in traditional multicast protocols, routers solely duplicate packets to the correct interfaces, with BIER they also actively modify packets so as to update, in-path, this set of destinations. Similarly to SR, the set of destinations for a packet

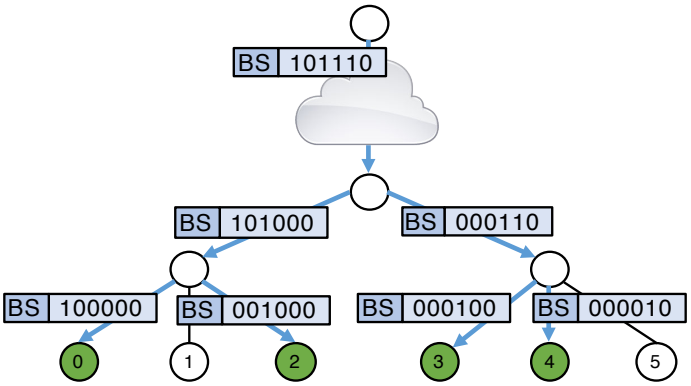


Figure 1.6 – Example of BIER bitstring processing

Algorithm 2 Bitstring Processing at BIER Router

$p \leftarrow$ incoming packet
$B \leftarrow p.bitstring$
$I \leftarrow \emptyset$ ▷ *set of outgoing interfaces for this packet*
$D \leftarrow [\emptyset, \ldots, \emptyset]$ ▷ *map from outgoing interface to bitstring*
for each $b \in B$ **do**
  $i \leftarrow$ interface leading to b (unicast FIB lookup)
  $I \leftarrow I \cup \{i\}$
  $D[i] \leftarrow D[i] \cup \{b\}$
end for
for each $i \in I$ **do**
  $p' \leftarrow p.copy()$
  $p'.bitstring \leftarrow D[i]$
  send p' over interface i
end for

is specified by the source. Alternatively, and also similarly to SR, this set can be specified by the first BIER router traversed by the packet, if BIER is to be provided within a domain to which the source does not belong.

The key component to allow such capability lies in agreeing (in advance, and out-of-band) upon a mapping between each node (identified by its IP address) and an integer index. The second key component is the embedding, within each packet, of a *bitstring* representing the set of destinations for that packet. More precisely, bit i in the bitstring of a packet is set if and only if node i is part of the set of destinations for that packet. Upon sending a packet, a source will embed a bitstring with the set of destinations for that packet, therefore solely taking care of group management. Whereas with PIM, group management happens in-band (with "join" messages) and involves the routers recording state, group management with BIER is thus completely out-of-band.

BIER requires that, as an invariant, the bitstring always contains the current destination set for a given packet. That is, a packet starts with a bitstring containing the whole intended set for the packet, and upon reaching a destination only contains the bit corresponding to that destination. To achieve this, BIER routers use a simple mechanism, described in algorithm 2 and illustrated in figure 1.6. For each destination contained in a given bitstring, unicast routing tables are used to determine which interface can be used to reach (via the shortest path) that destination. Then, the packet is replicated to each of those interfaces leading to at least one destination. While sending the packet over a given interface, its bitstring is updated so that it contains only those destinations reachable via that interface.

In order to optimize the implementation, it is possible to build a special table containing a mapping between interfaces and bit-masks of destinations reachable via this interface. Then, an incoming bitstring is simply AND'd with each of these bit-masks to determine the outgoing bitstring that should be sent over the corresponding interface – when this results in an empty bitstring, the packet is not forwarded over that interface.

It is important to note that the way in which node IP addresses are mapped to BIER identifiers

is not algorithmically significant, as long as the mapping is globally distributed and agreed upon between nodes. An interesting way of enforcing a mapping without having to distribute it is, for example, to use IPv6 destinations addresses in which one pre-defined byte carries the BIER identifier.

Similarly, the way the bitstring is encoded in packets does not influence the functioning of the protocol. Notably, the IETF has defined two simple encapsulation mechanisms [68]: (i) an MPLS header and (ii) a UDP header. The UDP header is arguably more flexible, as it allows BIER packets to be carried over the Internet seamlessly, by establishing a point-to-point tunnel between two successive BIER routers. In addition, a proposal has been made to carry the bitstring within an IPv6 extension header [69]. Finally, an interesting proposal [70] consists of encoding bitstrings directly within IPv6 addresses. For simplicity, and for consistency with work carried on SR, the encapsulation mechanism that has been retained throughout this manuscript consists of a custom IPv6 hop-by-hop extension header carrying the bitstring of BIER packets, with the destination address being a fixed multicast address.

Related Work

In [71], a proposal to increase the resiliency of multicast distribution with BIER or BIER-TE[8] with fast re-route mechanisms is introduced. This is achieved by the encoding of an additional *backup path* in the BIER-TE header, and of a *reset bitmask* to be applied after taking the backup path in order to avoid duplicate packets. A similar mechanism is introduced in [72], where alternative paths are explored with BIER when links fail. In [73], an SDN controller is implemented and evaluated, which can configure an OVS BIER router through the OpenFlow protocol. Compared to a standard SDN multicast solution, this approach only requires configuring the ingress router of the BIER domain, by mapping destination multicast addresses to BIER bitstrings. The use of this implementation over an optical network testbed is demonstrated in [74]. In [75], use of BIER-TE over a time-slotted low-power wireless network is demonstrated. Inspection of the bitstring at the egress node facilitates monitoring transmission failures. Finally, a data-plane implementation of BIER in P4 for programmable hardware is presented in [76], underlining the simplicity of packet processing allowed by BIER.

1.3 Thesis Statement: Augmenting Data Centers through the Network Layer

Section 1.1 has illustrated that data center architectures rely on complex architectures in order to address scalability issues. Notably, section 1.1.1 has demonstrated how complex tunneling architectures are often used as a resort to provide tenant isolation. Similarly, section 1.1.2 has shown that deployed data center topologies are often complex, with a need for custom addressing mechanisms and/or routing protocols, with machines sometimes involved in the routing plane. Then, section 1.2 has introduced two network architectures (SR and BIER) which add flexibility to the Layer-3. A shared property of these two architectures is that they provide source-routing-inspired services, wherein paths to be followed by, set of destinations to be reached by, or set of instructions to be applied on a packet are instructed by the source of that packet – or, by the first node of an administrative domain.

Another property shared by both network architectures introduced in section 1.2 is that they can be implemented on top of IPv6 by using extension headers. The flexibility provided by those architectures, as well as the simplicity of running them atop an IPv6 network, make them interesting candidates for augmenting data centers (*i.e.,* providing services at the network layer) while providing a simpler architectural model (*i.e.,* avoiding complex overlays by using only a flat addressing space).

In itself, IPv6 possesses properties making it a good candidate for providing a large-scale data center infrastructure.

- First, and most obviously, the large addressing space provided by IPv6 (with 2^{128} addresses as compared to 2^{32} addresses with IPv4) is necessary to host large-scale data centers. For instance, a data center using public IPv4 addressing with a `/16` IPv4 prefix could only host $2^{16} = 65536$ virtual machines.

[8] BIER-Traffic Engineering, encoding not destinations but link adjacencies to be traversed by a packet.

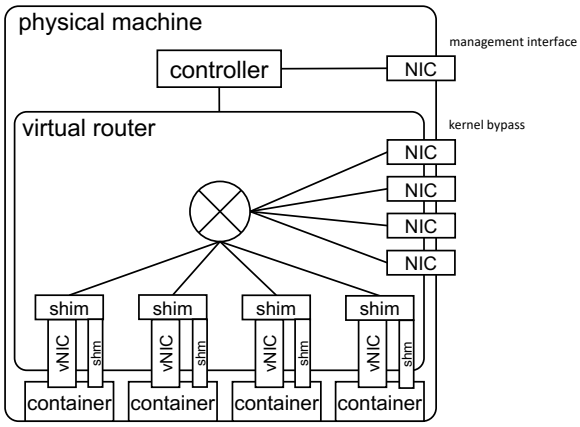


Figure 1.7 – Example of shim-layer running in a virtual router. In that example, workloads are containers, and a shared memory (`shm`) is used so that the shim layer can transparently retrieve state from the application (as in part IV).

- Second, this large addressing space can be leveraged to provide a *flat* architectural model, without complex overlays. By using the principle of locator-identifier separation [77], it is indeed possible to provision only a *physical* addressing scheme, where the routing infrastructure only takes care of the reachability of physical machines. On top of that, tasks (VMs, containers, *etc.*) are assigned *virtual* identifiers, decoupled of their physical location. Note that such a similar architecture running atop IPv4, VL2 [39], had been discussed discussed in section 1.1.2. With IPv6, a possible instantiation of this principle is Identifer-Locator Addressing (ILA) [78]. ILA consists of splitting the IPv6 space into two parts, and using the 64 low-order bits of addresses as identifiers, with the 64 low-order bits of addresses as locators. Naturally, the routing infrastructure will use longest-prefix-match routing and therefore route packets to the machine corresponding to the locator. However, applications only communicate with a common pseudo-locator, which is then translated by a shim layer to the real locator hosting the identifier. This way, tasks can be migrated without the applications noticing.

Those two reasons argue in favor of using IPv6 data centers. Nonetheless, I argue that it is only with the help of in-Layer-3 services, that one can make the case for flexible *and* efficient IPv6 data centers. This can be achieved by enriching the Layer-3 (through *e.g.,* SR or BIER), adding custom behavior in the network stack by *transparently connecting* it to the applications, while letting these applications only treat native IPv6 packets.

To achieve such a *transparent connection* between the network stack and the applications, some assumptions must be made. Notably, as in [39], machines are assumed to be assigned fixed ("physical") addresses, with the routing infrastructure of the data center able to provide connectivity between all machines. Only upon topology changes (adding or removing of machines and/or routers) will the routing infrastructure be updated by the routing protocol running in the data center. Then, each machine hosts several VMs/containers, and a virtual router dispatches packets between the physical interfaces of the machine and the virtual interfaces bound to the hosted VMs/containers. Then, *transparently connecting* the Layer-3 stack to the applications is done by way of a *shim layer* running in that virtual router (figure 1.7), which will be able to pre-process packets *before* handling them to the application (or post-process packets after receiving them from the application). For certain scenarios, a crucial property of this shim layer is that, running in a privileged virtual router, it can *inspect* the state of the applications, without interacting with them. This way, unmodified application can benefit from additional services, transparently provided by the network layer.

Examples of pre-processing operations which can be performed within such a shim layer, and which are explored throughout this manuscript, are:

- inspecting whether a VM has completed migration, and deciding accordingly whether to forward locally or remotely – allowing transparent VM migration (part II);

- detecting whether a multicast packet is received out-of-order, and if so requesting suitable retransmissions, before forwarding packets in-order to the application – enabling reliable multicast (part III);
- detecting whether a host has a cached copy of a multicast packet and can perform a retransmission of it, or whether the retransmission request should be forwarded further – enabling peer-based multicast recoveries (part III);
- inspecting whether an application is "available" to serve a query, before forwarding the corresponding packet to that application or rather forward it to another one – enabling load-aware load-balancing (part IV);
- inspecting whether an application has Layer-4 state for a connection, and deciding whether to forward packet to that applications or rather to another one – enabling load-balancing consistency (part IV);
- inspecting whether an application is overloaded, and if so, requesting that another instance is launched – enabling monitor-less autoscaling (part IV).

In sum, such an augmented Layer-3 allows offering task mobility (part II), reliable multicast (part III), and load-balancing with autoscaling (part IV) as a network service, while using unmodified applications. As will be studied throughout this manuscript, this has benefits in terms of network traffic overhead, optimization of resource utilization, quality and fairness of service, and energy reduction.

1.4 Definitions and Working Assumptions

This section briefly introduces the working assumptions that will be used throughout this manuscript, and defines terms and notations which will be used recurrently.

- A *router* is a hardware device which performs Layer-2 or Layer-3 forwarding between interfaces according to manually or automatically configured tables. Note that the term *router* will be used to refer to any forwarding device, irrespective of whether it performs Layer-2 or Layer-3 forwarding (whereas devices capable of Layer-2 forwarding, and by extension devices capable of both Layer-2 and Layer-3 forwarding, are sometimes referred to as *switches* in the literature).
- A *machine* is a physical computer, running a single operating system and connected to a network via physical network cards (NICs). It can itself host virtual appliances (VMs or containers) by way of a hypervisor – and these virtual appliances themselves are connected to the host via virtual network interfaces. Throughout this manuscript, machines will often be represented by the letter m. Machines will sometimes be referred to as *hosts* (due to their ability to host these virtual appliances).
- An *application instance* is a piece of software running as a Virtual Machine or a container, providing a service to consumers that would want to interact with the application. This is usually achieved by running a server software, *e.g.,* an HTTP server. Application instances can be replicated for scaling purposes, in which case it is assumed that each of the instances can provide identical services for a client – it is however necessary to ensure that a single network flow is always handled by the same application instance. By metonymy with the software running within them, application instances will occasionally be referred to as *servers* throughout this thesis. They will also be referred to as *tasks* or *workloads*, when this is more appropriate as per the context. Throughout this thesis, application instances will be represented by the letter i (in chapter 4), v (in chapter 3, where v is a mnemonic for VIP), or s (in part IV, where s is a mnemonic for server).
- A *rack* is a set of machines, located in a single physical pod, and connected to a Top-of-Rack (ToR) router.
- A *virtual router* is a softwarized router running in a physical machine, and whose purpose is to provide inter-connection between the physical interfaces of the machine, and the virtual interfaces bound to the application instances hosted by that machine.

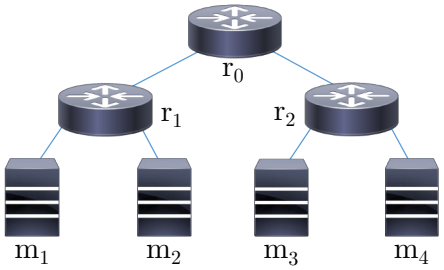


Figure 1.8 – Toy network example

- A *data center* is defined to be a single entity, containing a set of *routers* and *machines*, and connected to the outside via a *border gateway*. Within the data center, an (IPv6) physical addressing plane is used by the machines, *i.e.,* each machine is assigned a *Physical IP address* (PIP). Note that such PIPs do not necessarily have to be globally routable. A routing protocol is assumed to have converged, so that machines can communicate with one another transparently within the data center. Application instances are addressed by *Virtual IP addresses* (VIPs), and the VIPs are advertised by the border gateway of the data center, to the remainder of the Internet. The border gateway is in charge of routing packets coming from the outside of the data center and addressed to a VIP, to the correct corresponding machine(s). To do so, it will encapsulate these packets and route them to the PIP of that machine. The virtual router running there will then be in charge of decapsulation and handing over to the correct application instance, and of any extraneous behavior as defined in this thesis.

With these definitions, it is possible to provide a very simple model of a data center network. More precisely, let R be the set of routers in the data center and M be the set of physical machines. Then, the data center network can be modeled as a directed graph $G = (V, A)$, where $V = M \cup R$ is the set of vertices, and $A \subseteq (M \cup R)^2$ the set of arcs. Existence of an arc $(u, v) \in A$ models the existence of a physical link between nodes u and v. Such links can exist between two routers, between a machine and a router, but also between a machine and itself (thereby modeling a loopback interface).

Then, for each ordered pair of machines $(m, m') \in M^2$, a list $A_{mm'} \in 2^A$ represents the path from m to m'. For example, given the topology depicted in Figure 1.8 with three routers r_0, r_1, r_2 and four machines m_1, m_2, m_3, m_4, the arcs of the graph will be as follows: $A = \{(m_1, m_1), (m_1, r_1), (r_1, m_1), (m_2, m_2), (m_2, r_1), (r_1, m_2), (m_3, m_3), (m_3, r_2), (r_2, m_3), (m_4, m_4), (m_4, r_2), (r_2, m_4), (r_1, r_0), (r_0, r_1), (r_2, r_0), (r_0, r_2)\}$. The path from *e.g.,* m_1 to m_3 will be $A_{m_1 m_3} = \{(m_1, r_1), (r_1, r_0), (r_0, r_2), (r_2, m_3)\}$.

Finally, generic mathematical notation used throughout this thesis is introduced in table 1.1.

Notation	Definition
E^2	Cartesian product $E \times E = \{(x, y) : x \in E, y \in E\}$
2^E	Set of parts of E $\{F : F \subseteq E\}$
$\mathbb{N}$	Set of non-negative integers $\{0, 1, \ldots\}$
$\mathbb{R}$	Set of real numbers
$\mathbb{R}^+$	Set of non-negative real numbers $\{x \in \mathbb{R} : x \geq 0\}$
$\mathbb{C}$	Set of complex numbers $\{x + iy : (x, y) \in \mathbb{R}^2\}, i^2 = -1$
$\lfloor x \rfloor$	Floor function ($\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$)
$\lceil x \rceil$	Ceiling function ($\lceil x \rceil - 1 < x \leq \lceil x \rceil$)
$\log x$	Natural logarithm (in base e)
$\log_2 x$	Logarithm in base 2
$\mathbf{P}[X]$	Probability of event X
$\mathbf{P}[X\|Y]$	Probability of event X conditioned to Y
$\mathbf{E}[X]$	Expected value ("average") of random variable X $\mathbf{E}[X] = \begin{cases} \sum_x x \cdot \mathbf{P}[X = x] & \text{if } X \text{ is discrete} \\ \int_{\mathbb{R}} x f(x) \mathrm{d}x & \text{if } X \text{ has density } f \end{cases}$
CDF of X	Cumulative Distribution Function of random variable X $\mathrm{CDF}_X(x) = \mathbf{P}[X \leq x]$
$f(x) =_{x \to a} o(g(x))$	f is dominated by g $f(x) = \varepsilon(x)g(x)$ with $\lim_{x \to a} \varepsilon(x) = 0$
$f(x) \sim_{x \to a} g(x)$	f is equivalent to g $f(x) = (1 + o(1))g(x)$
$f(x) =_{x \to a} \mathcal{O}(g(x))$	f is bounded above by g $f(x) = \varepsilon(x)g(x)$ with $\varepsilon(x) \leq M, \forall x \in (x - \delta, x + \delta)$, for some $M > 0$ and $\delta > 0$
$f(x) =_{x \to a} \Theta(g(x))$	f is bounded above and below by g $f(x) = \varepsilon(x)g(x)$ with $m \leq \varepsilon(x) \leq M, \forall x \in (x - \delta, x + \delta)$, for some $m > 0, M > 0$ and $\delta > 0$

Table 1.1 – Generic mathematical notation

Chapter 2

Thesis Contributions

This chapter concludes this introductory part by providing a summary of contributions in section 2.1 and a list of publications in section 2.2.

2.1 Thesis Summary and Outline

This thesis studies the usage of network-layer protocols to provide task mobility, reliable content distribution, and load-balancing in data center networks. It comprises 5 parts and 10 chapters, structured as follows.

Part I provides an introductory discussion. In chapter 1, background on data-center architectures and associated network protocols is introduced. Then, two interesting Layer-3 architectures are introduced, SRv6 and BIER, which make use of source-routing paradigms in order to extend the network layer. Finally, a discussion is made about how using such architectures can help augment data centers by providing scalability primitives directly at the network layer. Chapter 2 then summarizes how this concept was applied throughout this thesis, to provide task mobility, reliable content delivery, and load-balancing.

Part II studies **task mobility** in data centers. In chapter 3 (published in [79]), the use of SRv6 to provide **zero-loss VM migration** is introduced. Traditional network protocols for VM migration rely on completely migrated VMs to signal their new location to a centralized directory, thus yielding a transient period during which packets destined to such VMs are lost. To alleviate this, chapter 3 proposes to pre-allocate, with SR, a loose "migration path" consisting of the old and new physical machines: this way, packets always reach the correct machine, whichever step the migration process is in. This path is provisioned by the control plane prior to initiating the actual migration, and until after the migration has completed. This is achieved by introducing two new SR functions: the first one checks (at the source host) whether the link to the VM is still up and forwards packets to the VM or to the second segment accordingly; the second one checks (at the destination host) whether the VM has fully restarted, and locally buffers or forwards packets to the VM as a result. Implementation and evaluation on a virtual router (VPP) show that it is indeed possible to migrate VMs without losing packets, thus decreasing latency and flow completion time for the hosted application.

Having introduced the possibility of task migration as a network service, chapter 4 (published in [80]) studies how **flow-aware workload migration** can be achieved, that is, migrating communicating task closer to each other so as to optimize network traffic. Traditional data center management architectures have been introduced that consider inter-task network demands, or the cost of workload migration, but not both. In chapter 4, a multi-objective optimization program is introduced, aiming at maximizing the total inter-task throughput while minimizing the number of migrated tasks and maximizing the number of newly allocated tasks. A Mixed Integer Non-Linear Programming (MILNP) formulation of the problem is introduced to capture the different constraints and objectives, and is linearized into a Mixed Non-Linear Programming (MILP) formulation. To compute the set of Pareto-optimal solutions, the ε-constraint method is used as a baseline. Through simulations, the proposed approach is demonstrated to efficiently increase the total achievable throughput by migrating some of the communicating tasks close to one another. Due to the high computational cost of this approach, a heuristic solution method is

proposed, which incrementally computes solutions of increasing migration cost, so as to return an approximate Pareto front. Simulations show that the proposed heuristic is able to decrease the computational solving time by up to two orders of magnitude, while providing solutions close to those which are optimal.

Part III studies **reliable content delivery** in data centers. In chapter 5 (published in [81]), the use of BIER to provide **efficient reliable multicast** is introduced. Traditional reliable multicast protocols use a Negative-ACKnowledgement (NACK) scheme, whereby lost packets are signaled by corresponding destinations to the source. Then, the source performs either unicast retransmissions to each of the failing destinations, or a multicast retransmission to the whole group. In chapter 5, a NACK-based protocol aiming at minimizing retransmission traffic is proposed, taking advantage of BIER. Instead of using per-flow bitstrings, different bitstrings are used *per-packet*: retransmission traffic for a packet is sent only to the set of those destinations that have missed that packet. To achieve this, the source collects NACKs emitted by different destinations during a short time window, so as to construct a suitable retransmission bitstring. The protocol is implemented in a network simulator (`ns3`), and packet-level simulations are conducted in different topologies with different loss patterns. Especially in cases where a subset of the data-center exhibits localized losses, traffic is shown to be reduced as compared to multicast-based retransmissions (because the whole tree does not have to be flooded) and to unicast-based retransmissions (because the bottleneck links do not have to carry multiple copies of retransmitted packets). To complete the analysis, a mathematical model quantifies the footprint of retransmission traffic for unicast-, multicast-, and BIER-based retransmission, in arbitrary tree topologies. A first-order approximation when $\alpha \to 0$ (with α the link loss rate) is derived in theorem 5.1, showing that retransmission traffic scales as $L \log L$ with BIER retransmissions, as compared to $L \log^2 L$ with unicast retransmissions and L^2 with multicast retransmissions (where L is the number of links in the tree), theoretically confirming those benefits.

In chapter 6 (submitted as [82]), the use of *reliable BIER* is extended to provide **peer-assisted multicast recoveries**. An extension to the BIER data-plane is proposed, which allows destinations to learn about topologically close peers that have subscribed to the same multicast flow. This is achieved by carrying and updating a *peerstring* in each packet, containing the set of destinations reached by the parent node for this packet. Then, upon missing a packet, SR is used by failing destinations to direct retransmission requests (NACKs) through a path comprising (i) one or several peers and (ii) the source, until finding one peer (or ultimately, the source) able to send a retransmission of the missed packet. Different policies for peer selection are proposed and mathematically evaluated. First, two simple static policies are analyzed, namely (i) selecting a random peer in a destination's subtree and (ii) selecting a designated peer in that subtree. Mathematical analysis reveals that the second policy generates less retransmission traffic (as retransmissions are sent by only one peer) but is less likely to succeed (since if that peer has not received the packet, no peer will be able to obtain a retransmission). Second, a dynamic policy is introduced, wherein each peer tries to dynamically learn from which peer it is more likely to obtain a retransmission. The proposed architecture is evaluated through packet-level network simulations, in different topologies and with different policies. Evaluation shows that it is able to reduce the load on core links (as compared to chapter 5) by increasing retransmission locality, thus decreasing the overall network overhead.

Part IV studies **load-balancing** in data centers. In chapter 7 (published in [83, 84]), the use of **SRv6 for application-aware load-balancing** is studied. Usually, load-balancing devices in data centers are replicated, for resiliency and scalability. In doing so, consistent hashing is used to ensure that flows are assigned to the same server, whichever load-balancer they go through, and even in case of reconfiguration of the pool of application instances. However, these approaches assign queries to application instances regardless of their current load. To handle this, chapter 7 introduces the use of SR to direct the first packet (TCP SYN) of a query through a chain of *two random candidate* instances, each one successively deciding whether or not to accept it based on its local state. This provides a load-aware load-balancing mechanism running within the network layer, and without monitoring. To avoid unnecessary triangular traffic, further packets are directed directly towards the instance having accepted the connection. This is achieved by using an in-band signaling protocol, where custom SR functions are used between the load-balancers and the servers to install and remove per-flow state. Resiliency is ensured by way of a consistent hashing algorithm assigning flows to lists (instead of singletons) of instances in a reliable way. A mathematical model of the client response time of the system for Poisson arrivals and exponential

service times is derived. Using this model, it is possible to quantify the benefits of the proposed approach in terms of expected response time, tail response time, server load fairness, and energy reduction. The analysis shows that, under the reasonable assumption that the network delay is smaller than the mean service time, the performance is always increased as compared to random single-choice load-balancing (theorem 7.1). The proposed architecture is implemented as a plugin for a virtual router (VPP), able to extract information from a standard HTTP server (Apache) without disruption through a shared memory channel, and using this information to take connection acceptance decisions. Evaluation on a 48-instance testbed with synthetic and real traffic confirms these benefits, notably showing that it is possible to provide the same average quality of service (client response time) by reducing the number of VMs by 17% as compared to single-choice random load-balancing.

In chapter 8 (published in [85]), the feasibility of implementing a **stateless load-aware load-balancing** architecture in hardware is explored. Implementing load-balancing dispatchers in hardware is desirable for efficiency and scalability reasons. However, this is incompatible with maintenance of per-flow state, as required by load-aware architectures. To circumvent this, chapter 8 explores the use of *covert channels* to carry such state within the packet headers. The same mechanism is used as in chapter 7 to establish connections – *i.e.,* connection establishment packets are sent through a chain of candidate instances until one of them accepts to handle the query, based on its local state. However, instead of installing flow state in the load-balancer, the instance having accepted the connection communicates its *position* in the SR header in a covert channel back to the client. This value is then automatically reflected by the client to the load-balancer, which can thus use this information to direct the packet to the correct server. Low-order bits of TCP timestamps are proposed as a covert channel usable by standard, unmodified TCP clients. In addition, the load-balancer ensures reliability in case of changes in the pool of instances by way of consistent hashing versioning. That is, packets are sent with an SR header containing instances that were previously the result of the consistent hashing operation. The feasibility of this architecture is demonstrated by implementation of a prototype on programmable hardware, using the P4 language and targetting the NetFPGA-SUME platform. Data-plane per-packet simulations of that implementation show that the incurred parsing latency is negligible (in the order of 10 μs). Simulations show that the proposed consistent hashing mechanism improves resiliency by one order of magnitude as compared to mechanisms that do not use versioning. In sum, this architecture allows to obtain the same benefits as those introduced in chapter 7, while providing a low-latency line-rate hardware implementation and increasing consistent hashing resiliency.

Chapter 9 (submitted as [86]) introduces a **monitor-less auto-scaling** architecture using SRv6. In order to meet variations in traffic demands while ensuring a predefined level of service, data centers use auto-scalers to, in real time, adapt the number of replicated instances of a given service. Auto-scalers rely on centralized architectures to make these decisions, thus incurring a monitoring overhead. In chapter 9, the use of SRv6 to provide joint monitor-less load-balancing and auto-scaling is studied. Application instances are ordered into a fixed chain, and new queries are sent along this chain until one of them accepts to serve the connection, using the data-plane introduced in chapter 7. In addition, the last instance of the chain monitors its own usage state to trigger down- or up-scaling of the chain. Indeed, the last instance of the chain receiving too few queries indicates that the chain is over-provisioned, whereas its receiving too many queries indicates that the chain must be upscaled. A Markov chain model of the performance of the system is derived, allowing to express the number of queries held by each server. Using the Recursive Renewal Reward (RRR) technique, it is possible to derive the corresponding expected client response time, in the case of small chains, and therefore to deduce the possible energy savings offered by the system (*i.e.,* the possible reduction in number of servers while achieving a similar service quality as with random load-balancing). Implementation of the architecture as a VPP plugin, and evaluation with real traffic traces confirms that these benefits hold in real environments: the same average client response time can be achieved with lesser VMs as with random load-balancing, while simultaneously diminishing the tail of the response time distribution.

Finally, part V concludes this manuscript, and a summary in French is provided in appendix A.

2.2 List of Publications

The following publications were published or submitted during the course of this PhD.

Journal Publications

- Yoann Desmouceaux, Sonia Toubaline, Thomas Clausen, *Flow-Aware Workload Migration in Data Centers*, Springer Journal of Network and Systems Management, vol. 26, no. 4, pp. 1034–1057, October 2018 (chapter 4).
- Yoann Desmouceaux, Thomas Clausen, Juan-Antonio Cordero-Fuertes, Mark Townsley, *Reliable Multicast with B.I.E.R.*, IEEE/KICS Journal of Communications and Networks, vol. 20, no. 2, pp. 182–197, April 2018 (chapter 5).
- Yoann Desmouceaux, Pierre Pfister, Jérôme Tollet, Mark Townsley, Thomas Clausen, *6LB: Scalable and Application-Aware Load Balancing with Segment Routing*, IEEE/ACM Transactions on Networking, vol. 26, no. 2, pp. 819–834, April 2018 (chapter 7).
- Yoann Desmouceaux, Thomas Clausen, Juan-Antonio Cordero-Fuertes, Mark Townsley, *Reliable BIER with Peer Caching*, submitted to IEEE Transactions on Network and Service Management (chapter 6).

Conference or Workshop Publications

- Yoann Desmouceaux, Marcel Enguehard, Victor Nguyen, Pierre Pfister, Wenqin Shao, Éric Vyncke, *A Content-aware Data-plane for Efficient and Scalable Video Delivery*, Proc. IEEE IM, April 2019, pp. 10–18 (not covered in this thesis).
- Yoann Desmouceaux, Mark Townsley, Thomas Clausen, *Zero-Loss Virtual Machine Migration in IPv6 Data-Centers with Segment Routing*, Proc. IEEE CNSM, 1st SR+SFC workshop, November 2018, pp. 420–425 (chapter 3).
- Benoît Pit--Claudel, Yoann Desmouceaux, Pierre Pfister, Mark Townsley, Thomas Clausen, *Stateless Load-Aware Load Balancing in P4*, Proc. IEEE ICNP, 1st P4WE Workshop, September 2018, pp. 418–423 (chapter 8).
- Yoann Desmouceaux, Pierre Pfister, Jérôme Tollet, Mark Townsley, Thomas Clausen, *SRLB: The Power of Choices in Load Balancing with Segment Routing*, Proc. IEEE ICDCS, June 2017, pp. 2011–2016 (chapter 7).
- Yoann Desmouceaux, Marcel Enguehard, Thomas Clausen, *Joint Monitorless Load-Balancing and Autoscaling for Zero-Wait-Time in Data Centers*, submission in preparation (chapter 9).

Other Contributors

Although this thesis reports work mostly conducted by myself during these three years of PhD, some of the work reported in this thesis results from a collaboration with my co-authors.

My colleagues Pierre Pfister and Jérôme Tollet provided ideas and discussions on chapters 7, and Pierre ran the experiments reported in figure 7.25. The experiments reported in chapter 8 are the result of a research internship conducted at Cisco by Benoît Pit--Claudel, and mentored by myself with input from Pierre Pfister.

Outside my direct work circle, Sonia Toubaline provided insightful input on the modeling part of chapter 4. Similarly, Juan-Antonio Cordero-Fuertes provided valuable input on the modeling part of chapters 5 and 6. Especially, the experiments reported in section 6.5.3 result from a collaboration with Juan-Antonio.

Data Availability

All the data and scripts necessary to reproduce the graphs included in this thesis can be publicly accessed, and can be reused by anyone interested in doing so[1].

[1] `https://github.com/yoannd/phdthesis-data`

Part II

Task Mobility

Chapter 3

Zero-Loss Virtual Machine Migration with IPv6 Segment Routing

Virtual Machine (VM) live migration [87] is a way to transfer a VM from a machine to another, by iteratively copying its memory. It is further suggested in [88, 89] that it is possible to perform live migration of individual processes or containers, thus attaining one further level of granularity of task placement flexibility. These techniques naturally find applications in the context of data centers, as these host heterogeneous workloads, whose lifetimes can vary greatly, and which can exhibit rapidly changing resource demands and inter-application dependencies. With live VM migration, applications need not be tied to a specific machine – and their relocation can become part of a "natural" mode of data center operation. In this context, architectures have been developed in which workload migration is used as a baseline to achieve different goals [90]: energy minimization [91], network usage minimization [92], operational cost minimization [93], maintenance [94], *etc.* Thus, VM migration not only provides a way to accommodate hardware failures without service downtime, but more generally may be beneficial to the efficiency of the whole data center.

From a network perspective, a challenge raised by live migration lies in maintaining connectivity to a VM *after* it has been migrated. Indeed, hypervisors normally assume that VMs are migrated within a single LAN, using Reverse ARP (RARP) to advertise the new location of a VM after migration. Traditional techniques to overcome this issue rely on Layer 2 overlays, such as VXLAN [25] or NVGRE [26]. Other approaches include making use of Mobile IP [95] or of LISP [77]. The emergence of IPv6 [14] data-centers introduces other opportunities, both for the addressing of workloads and for re-engineering the data-path. Among the proposed approaches for mobility within IPv6 data-centers, Identifier-Locator Addressing (ILA) [78] has been proposed at the IETF, using high-order bytes of addresses to denote locators and low-order bytes of addresses to denote identifiers.

A drawback of all these approaches is that they incur a period of time, during which packets addressed to the migrating VMs are lost. This raises concerns for applications that are intolerant to packet losses (*e.g.,* UDP-based delay-critical applications, virtual network functions, ...).

3.1 Statement of Purpose

The purpose of the chapter is to introduce a VM mobility solution, which provides "zero-loss" capabilities: assuming that the network is not lossy, any packet destined to a migrating VM will eventually be received by said VM. To that purpose, the Segment Routing (SR) architecture is leveraged.

As introduced in section 1.2.1, SR is an architecture which allows packets within a designated domain to be added an extraneous header, designating an ordered list of *segments* through which the packet is expected to go. Segments represent abstract functions to be performed on packets, and can be as simple as *forward to next segment* (enabling source routing and traffic engineering), but can also represent more complicate instructions (from custom encapsulation or routing behavior to complete virtual network functions). With IPv6 Segment Routing (SRv6), segments

are represented as IPv6 addresses embedded as a list in an IPv6 extension header [47].

The idea underlying this chapter is to use SRv6 to solve the locator/identifier mapping synchronization problem during VM migration, by letting the gateway (proactively) route packets destined to a migrating VM through a path comprising the old and the new host machine. This way, the responsibility of the old host machine is reduced to (i) forwarding packets locally while the migration is not complete and (ii) forwarding packets to the next segment (the new host machine) once migration has completed. This way, no packets are lost during the migration, whereas traditional solutions would require a mapping to be updated *once* migration has completed, leading to potential packet losses during this period of time. Furthermore, the overhead on the old host machine is kept to a minimum, since no tunnel has to be established and no Layer-2 overlay is required.

3.1.1 Related Work

Numerous solutions have been proposed to address the issue of maintaining Layer-3 network connectivity during and after VM migration. The simplest involves using a Layer-2 overlay [25,26], and relying on hypervisors sending gratuitous Reverse ARP messages after migration. In addition to the operational complexity incurred by such overlays, the VM is not reachable on the new host machine until the RARP has propagated, leading to potential packet losses.

Another simple solution consists of creating an IP tunnel between the source and the destination host machines. In [96], the Xen hypervisor is modified so that after migration, packets reaching the hypervisor at the source host machine are tunnelled towards the new host machine. After migration, the VM uses two different addresses (an "old" and a "new" address), and Dynamic DNS is used so that external clients can reach the VM via the new address. The drawback of this approach is that the hypervisor must co-operate with the destination host machine during an unpredictable amount of time by performing tunnelling. Furthermore, this is incompatible with para-virtualized interfaces such as *virtio-net* [97], where packets destined to VMs are not handled by the hypervisor.

In [98], Mobile IP is used to assist the migration process. Traffic from/to the VM is routed through a home agent, which tunnels it to the correct machine hosting the VM. It is the role of the hypervisor to update the home agent with the new location of the VM once it has migrated. Thus, after VM migration, packets can wrongfully reach the source host machine before registration of the new location is complete. This approach is improved in [99], by configuring, before migration, a dummy secondary interface for the destination network, and swapping primary and secondary interfaces after migration. This requires co-operation with the VM as two interfaces are used.

In [100], LISP is used to address the issue of Layer-3 connectivity during VM migration. Packets destined to a VM traverse a LISP router, which encapsulates them towards the current resource locator of the VM. The hypervisor is modified so that, after VM migration, the mapping system is updated to reflect the new resource locator of the VM. This avoids triangular routing, but once again the VM is not reachable during the period of time when the mapping is being updated.

Finally, other approaches orthogonal to the context of this chapter are worth mentioning. In [101], TCP options are used to facilitate migration of TCP connections. In a Software Defined Networking (SDN) context, [102] proposes a framework for virtual router migration, in which control planes and data planes are migrated during two distinct phases.

3.1.2 Chapter Outline

The remainder of this chapter is organized as follows. Section 3.2 introduces the high-level assumptions and mechanisms used for SR-based migration, before a formal specification is given in section 3.3. The proposed mechanism is then evaluated on different workloads in section 3.4. Finally, section 3.5 concludes this chapter.

3.2 SR-based Migration

As described in section 1.4, this chapter assumes that a VM is located within an IPv6 data-center, and accessible through a *virtual IP address* (VIP). Each machine within the data-center is accessible at a *physical IP address* (PIP), and can host several VMs. Located at the edge of the data-center, a gateway advertises (*e.g.,* with BGP) the whole virtual address prefix.

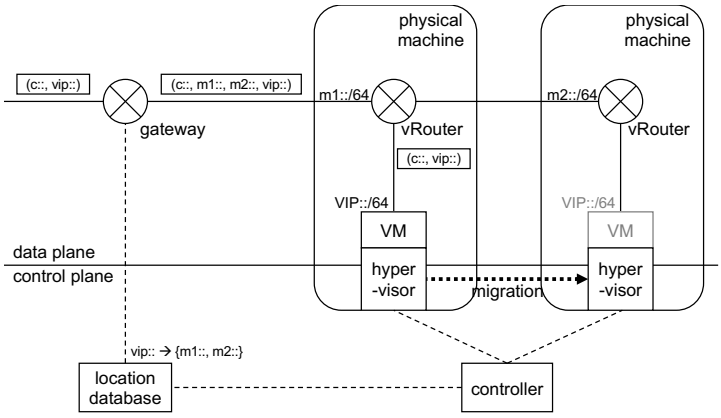


Figure 3.1 – SR VM Migration: migrating VM.

A *location database* maintains a mapping between each virtual address and the physical address of the machine hosting the corresponding VM. When receiving traffic for a given VIP, the router will apply an SR policy consisting of one segment (the PIP of the machine hosting the VM). Each machine is running a virtual router (in the implementation described in section 3.4, VPP[1]), which is connected to the physical interface(s) of the machine, and to each VM via a virtual interface (in the implementation described in section 3.4, a `vhost-user` interface[2]). In sum, under normal operation (when the VM is running on a single machine), the gateway will tunnel traffic for a VM towards the machine hosting it.

The mechanism introduced in this chapter assumes that an orchestrator is running in the data-center, which decides on the allocation of VMs to physical machines. When the orchestrator decides to move a VM from a host machine to another, it proactively modifies the routing tables of the gateway, so that the traffic for the corresponding VIP is applied an SR policy consisting of two segments, corresponding to the old and new machines hosting the VM. This way, when the VM is migrating, the gateway will direct traffic to the old host machine. As long as the VM has not completed migration, traffic will be intercepted by the old host machine (figure 3.1); as soon as migration is complete, the old host machine will simply forward traffic to the next segment (*i.e.,* the new host machine). This way, no synchronization is required between the host machines and the networking infrastructure: rather, traffic is loosely sent to a logical path comprising both machines.

3.3 Detailed Specification

This section introduces a formal description of the SR functions necessary to perform zero-loss migration, as well as the behavior of the gateway.

3.3.1 Definitions

Forward to Local (fw)

The fw function simply forwards the packet to the next segment. In SR terminology [50], this corresponding to the `END` behavior.

Forward to Local if Present (fwp)

The fwp function forwards the packet to the last segment (skipping intermediary segments), only if the corresponding VIP v is present locally; otherwise it acts as the fw function and forwards the packet to the next segment. The forwarding decision is made by the virtual router by inspecting its routing table, and seeing whether the entry for v corresponds to a virtual (local) interface, whose

[1] `https://gerrit.fd.io/r/vpp`
[2] `https://github.com/qemu/qemu/blob/master/docs/interop/vhost-user.txt`

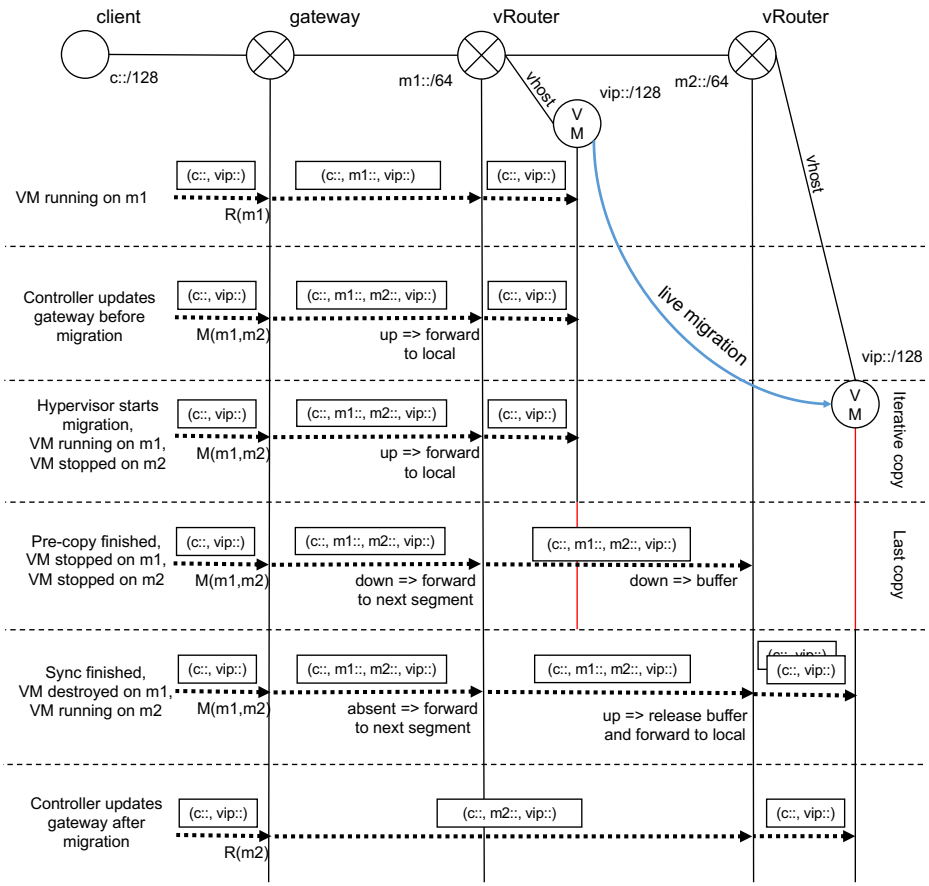


Figure 3.2 – SR Migration: detailed example.

link-status is up. This corresponds to the `END.S` behavior [50] with a custom policy for forwarding decisions.

Buffer and Forward to Local (bfw)

The bfw function inspects the last segment v. If v is not present locally (*i.e.,* if the corresponding virtual interface is not ready), it will buffer the packets for further delivery. Otherwise, it flushes any buffered packet to the virtual interface corresponding to v, and forwards the current packet to the last segment v. Implementation-wise, packets are buffered in the local memory of the virtual router, per interface and in order. Such packet buffers must be provisioned with a size large enough to handle all potential packets coming during the VM downtime phase. If a VM is expected to receive traffic at rate of r packets/s and to be down during Δt seconds, buffers must be provisioned with a size of $r \cdot \Delta t$ packets.

3.3.2 Detailed Migration Process

For each VIP v, the controller maintains an entry $L(v)$ in the gateway which is either $R(m_1)$ ("running on m_1") or $M(m_1, m_2)$ ("migrating from m_1 to m_2"), depending on the state of v. Conceptually, this forms a mapping $v \mapsto L(v)$, where $L(v)$ is the location of v. Practically, this is implemented by way of routing adjacencies in the FIB of the gateway. The gateway uses the *transit behavior* `T.INSERT` [50] for v, that is, this routing adjacency triggers insertion of an SR header on packets destined to v. When the gateway receives a packet for v:

1. If the corresponding entry $L(v)$ is $R(m_1)$, then a SR header $(m_1 :: fw, v)$ is inserted;
2. If the corresponding entry $L(v)$ is $M(m_1, m_2)$, then a SR header $(m_1 :: fwp, m_2 :: bfw, v)$ is inserted.

Experiment	Variant	Downtime	Lost/buffered packets
HTTP static	Non-SR migration	101 ms	1160
HTTP static	SR migration (drop)	107 ms	197
HTTP static	SR migration (buffer)	104 ms	216
HTTP dynamic	Non-SR migration	100 ms	1100
HTTP dynamic	SR migration (drop)	102 ms	190
HTTP dynamic	SR migration (buffer)	103 ms	199
iperf	Non-SR migration	192 ms	1090
iperf	SR migration (drop)	192 ms	859
iperf	SR migration (buffer)	185 ms	789
iperf sink	Non-SR migration	167 ms	3390
iperf sink	SR migration (drop)	163 ms	2810
iperf sink	SR migration (buffer)	163 ms	3050

Table 3.1 – Evaluation of SR migration: average VM downtimes and number of lost packets (or buffered packets for the corresponding mechanism) for the different evaluated workloads and migration mechanisms.

Normal Operation

Under "normal operation", that is, when a VM v is running on a machine m_1, the gateway is configured to map v to $L(v) = R(m_1)$. Thus, packets destined to v are forwarded to m_1 by way of a single-hop SR tunnel.

Migration Operation

Figure 3.2 examplifies the operation that occurs when a VM is migrated. When the controller decides to move v from m_1 to m_2, it updates the gateway so as to remap the entry for v to $L(v) = M(m_1, m_2)$. It then queries the hypervisors running on m_1 and m_2 to initiate the live migration process.

First, the VM is stopped on m_2 and running on m_1, while the memory of v is iteratively copied by the hypervisor (out-of-band) from m_1 to m_2. The gateway inserts an SR header $(m_1 :: fwp, m_2 :: bfw, v)$ in the packets it receives for v. Since the virtual router on m_1 sees that the virtual interface of the VM is up, upon triggering of the fwp function, it will simply forward these packets to that interface (skipping the m_2 segment).

Second, the hypervisor has finished iteratively copying the memory of the VM to the new host machine, and needs to perform the copy of the last memory pages as well as the CPU state. At this point, the VM is stopped on both m_1 and m_2. The gateway still inserts an SRH $(m_1 :: fwp, m_2 :: bfw, v)$ in packets it receives. Now that the VM is stopped on m_1 (which the virtual router at m_1 detects by inspecting the status of the corresponding virtual interface, now in link-down state), the fwp function will simply forward packets to the next segment. Since m_2 is not yet running the VM (which the virtual router at m_2 detects by inspecting the status of the corresponding virtual interface, also in link-down state), the bfw function will temporarily buffer the packet.

Third, the VM is started back on machine m_2. Before the controller (and thus the gateway) is notified of this, the gateway has a stale mapping $L(v) = M(m_1, m_2)$, and the same SRH is inserted in the packets destined to v. Thus, the first machine m_1 forwards these to m_2. Now, the bfw function on m_2 will trigger release of the buffered packets to the virtual interface of the VM, and further packets are forwarded without buffering.

Finally, the controller is notified of the end of the migration, and can update the gateway to map v to $L(v) = R(m_2)$: normal operation resumes, packets reach m_2 directly.

3.4 Evaluation

The SR functions described in section 3.3.2 have been implemented as VPP plugins. No collaboration between VPP and the hypervisor is needed; rather, forwarding decisions are made from within VPP by inspecting the state of the `vhost-user` interface corresponding to the VM of interest. Buffers for the bfw function are sized to 8192 packets (consuming 16 MB of RAM), allowing to sustain MTU-sized traffic at ≈ 1 Gbps for a typical VM downtime of 100 ms.

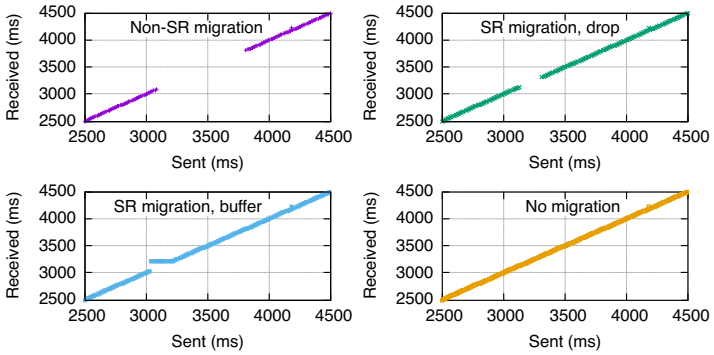


Figure 3.3 – Illustration of SR migration: *ping* experiment for the different mechanisms.

A simple testbed is set up with three physical machines, as in figure 3.1: the first plays the role of a gateway gw, and the two other ones m_1, m_2 represent compute nodes, which are candidates to host virtual machines. A client VM v_1 (representing the "outside" of the data-center) is attached to the gateway. A VM v_0 represents a server that will be migrated. That VM is first running on m_1, then migrated from m_1 to m_2. In order to reflect this, the gateway is configured in the $M(m_1, m_2)$ state, *i.e.,* it inserts $(m_1 :: fwp, m_2 :: bfw, v_0)$ in all packets destined to v_0. Four mechanisms are compared:

1. no-migration, *i.e.,* the VM is not migrated;
2. non-SR migration ("baseline scenario"), *i.e.,* the gateway is first in state $R(m_1)$, and once the migration is complete the controller puts the gateway in state $R(m_2)$ (without going through a migration state $M(m_1, m_2)$);
3. SR migration without buffer, *i.e.,* the gateway is in state $M(m_1, m_2)$ but packets received on m_2 while the VM is not yet up are dropped instead of buffered;
4. SR migration with buffer ("zero-loss migration" as in section 3.3.2), *i.e.,* the gateway is in state $M(m_1, m_2)$ and packets received on m_2 while the VM is not yet up are buffered.

The baseline scenario serves as an illustration of mechanisms such as LISP-based migration [100], wherein packet loss occurs due to the locator mapping being updated only after migration.

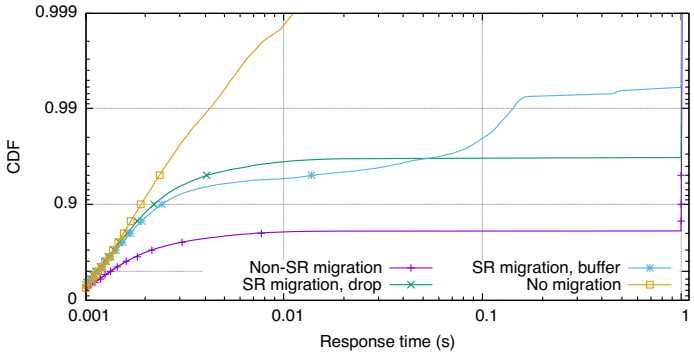
3.4.1 Ping (illustration)

In order to illustrate the behavior of these four mechanisms, a *ping*[3] is run between the client VM v_1 and the server VM v_0, while v_0 is migrated. One packet is sent every millisecond, and the RTT for each packet is recorded. Figure 3.3 shows the time at which an echo answer is received as a function of when the corresponding echo request was sent. Without migration, each answer is received approximately 0.1 ms after the corresponding query is sent. With *non-SR migration*, packets were lost during 722 ms (corresponding to the VM downtime, plus the network reconfiguration time), whereas with *SR migration without buffer*, packets were lost for only 174 ms (corresponding to the VM downtime). Finally, with *SR migration with buffer*, 175 packets were buffered while the VM was down, and replied to as soon as the VM went up again.

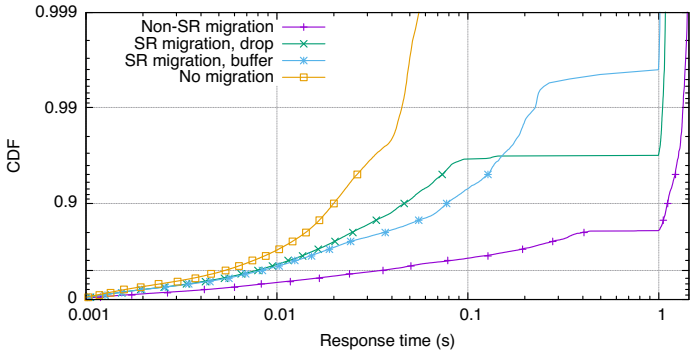
3.4.2 HTTP Workload

To understand the behavior of SR-migration when facing a delay-sensitive workload, a simple evaluation scenario is carried out. The server VM v_0 is set up with an Apache HTTP server – serving a default static file (whose size is 12 KB). As previously, the VM is first running on m_1, then migrated from m_1 to m_2. A traffic generator is attached to the gateway, sending a Poisson stream of 6000 queries with rate $\lambda = 1500\ s^{-1}$, during which the VM is migrated. The experiment is

[3] In its standard implementation, the `ping6` utility adapts its sending rate if it sees that probes are not replied to. For the purpose of these experiments, `ping6` was recompiled so that packets are sent with the same rate, no matter how many of them are not answered.



(a) Static HTTP workload



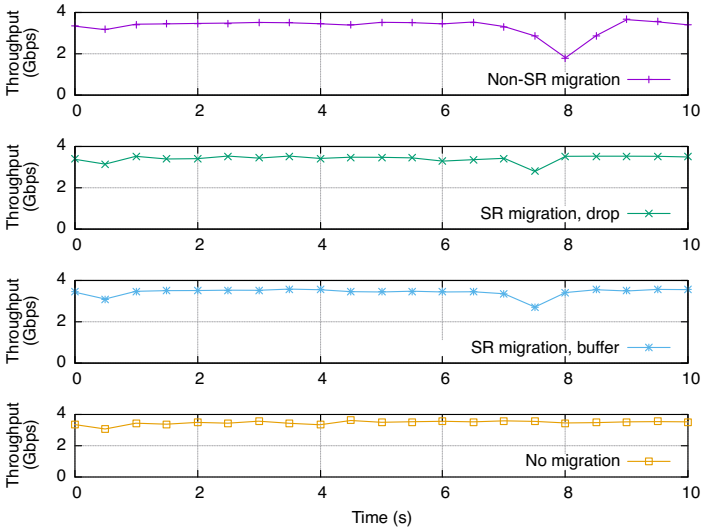
(b) Dynamic HTTP workload

Figure 3.4 – Evaluation of SR migration: response times for the HTTP workloads, $\lambda = 1500\ s^{-1}$ (log-log scales).

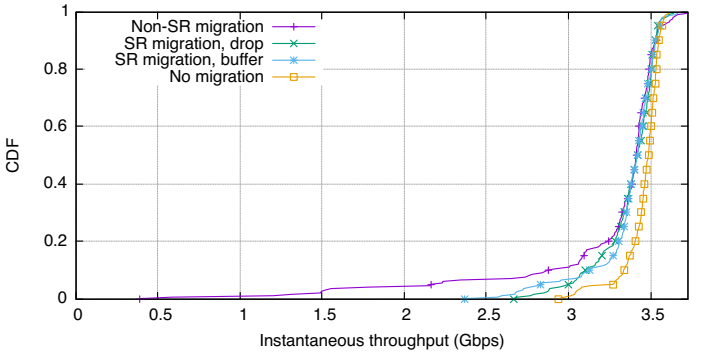
repeated 10 times, for each of the four previously-introduced evaluation scenarios. VM downtimes and number of lost or buffered packets (for this set of experiments, and all other experiments of section 3.4) are reported in table 3.1. Response times for all queries are recorded and reported in figure 3.4a.

The majority of queries are executed when the VM is not migrating, and exhibit small response times (≤ 3 ms). The other queries correspond to those that started while the VM was migrating. With *non-SR migration*, TCP SYN packets are lost when the VM is down, but some are also lost after the VM has restarted on m_2, due to the reconfiguration delay. Due to the SYN retransmit delay of 1 s, those queries exhibit a response time ≥ 1 s (more than 19% of queries are concerned). With *SR migration without buffer*, TCP SYN packets are lost when the VM is down, but as soon as the VM is up they are successfully transmitted again. This explains why queries experiencing a SYN retransmit are less numerous than with the baseline scenario: less than 4% of queries have a response time greater than 1 s. Finally, with *SR migration with buffer*, TCP SYN packets are buffered while the VM is down. The corresponding response time is simply delayed by the VM downtime (≈ 100 ms in these experiments), rather by the SYN retransmit delay. Less than 0.7% of queries exhibit a response time greater than 1 s, whereas more than 99.2% of queries have a response time lower than 200 ms.

A similar experiment is performed on a more realistic workload, consisting of drawing a random number k (from an exponential distribution with mean $\mathbf{E}[k] = 4$) and serving k copies of the previous static file. This allows to induce variability in the response times, as well as more network traffic. Again, the experiment is repeated 10 times, and client response times are recorded: results are depicted in figure 3.4b. With *non-SR migration*, more than 19% of queries are replied to within 1 s or more. In comparison, less than 4% of queries exhibit a response time greater than 1 s with *SR migration without buffer*, and this drops to 0.4% for *SR migration with buffer*. Furthermore, 99.5% of queries are served within less than 200 ms with the latter mechanism.



(a) Instantaneous aggregate throughput for one of the 10 runs



(b) CDF of the instantaneous aggregate throughput over the 10 runs

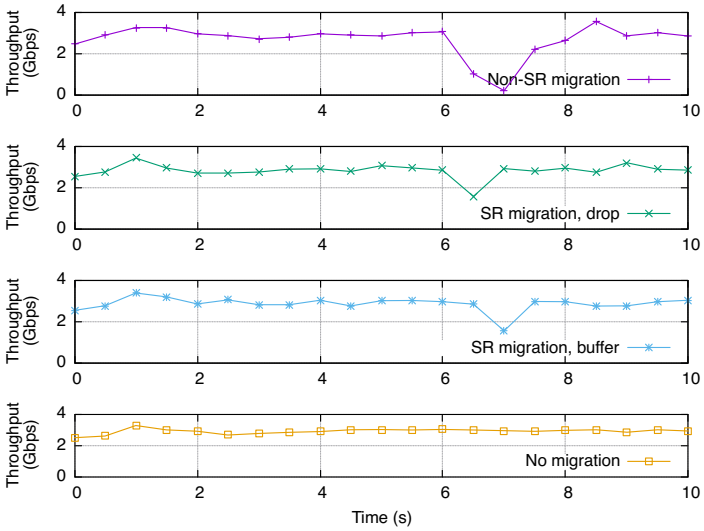
Figure 3.5 – Evaluation of SR migration: *iperf* experiments with 50 clients

3.4.3 Iperf Workload

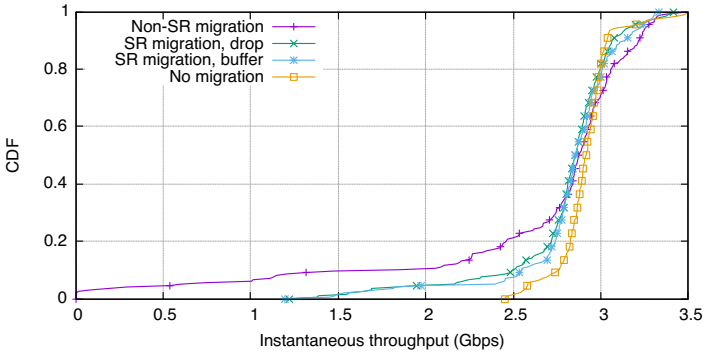
The previous set of experiments gave insight on the behavior of the different migration mechanisms in presence of short-lived flows. To understand the influence of SR-migration on longer connections, an experiment with throughput-intensive parallel TCP flows is performed. The same experimental platform as in the previous section is used – except that this time, a delay of 2.5 ms is added between the VM v_1 (representing clients) and the VM v_0 (representing the server to be migrated), so as to represent clients outside the data center. An instance of *iperf* with 50 parallel connections is started on the client VM, for 10 seconds (the rationale for using many parallel connections is to smooth the effects of TCP congestion control). Meanwhile, the server VM is migrated from m_1 to m_2, using the same 4 mechanisms as before. The experiment is repeated 10 times.

Figure 3.5a depicts the instantaneous throughput (aggregated over the 50 subflows) for one of the 10 runs, collected every 0.5 s. Before and during VM migration, the throughput is rather stable, oscillating around 3.5 Gbps. When the migration ends, around $t = 7$ s, there is a drop in throughput, due to the VM downtime ($\approx$ 150 ms in these experiments). While this drop is significant with *non-SR migration* (going to 1.8 Gbps in the run reported in figure 3.5a), it remains reasonably high with *SR migration* (2.7 Gbps in this experiment). The low performance of *non-SR migration* can be explained by TCP reducing its congestion window, due to the large amount of time when no packets are received by the VM. With *SR migration*, this downtime is lower, and thus the congestion window is less reduced. The impact of buffering, versus dropping packets while the VM is not yet up on the new host machine, seems to be negligible.

In order to quantify these behaviors, figure 3.5b depicts the distribution of the instantaneous aggregate throughput (collected every 0.5 s), over all the 10 runs. With the *no migration* sce-



(a) Instantaneous aggregate throughput for one of the 10 runs



(b) CDF of the instantaneous aggregate throughput over the 10 runs

Figure 3.6 – Evaluation of SR migration: *iperf sink* experiments with 50 clients

nario, the instantaneous throughput stays between 2.9 and 3.7 Gbps. *SR migration* is able to maintain the throughput above 2.3 Gbps at all times, whereas with *non-SR migration* the instantaneous throughput can drop to 0.4 Gbps during the migration phase. With *non-SR migration*, the throughput is lower than 2.2 Gbps for 5% of the time (compared to never with *SR migration*), and lower than 2.9 Gbps for 10% of the time (compared to 5% of the time with *SR migration*).

3.4.4 Iperf Sink Workload

The same set of experiments is repeated, except that in this case the *iperf* instance running in the VM is acting as a sink. This represents, *e.g.,* use-cases where the VM receives a lot of data – for instance, if it is an HTTP proxy or a firewall. In such a scenario, packets lost during the migration process are actual data packets (rather than simple TCP acknowledgements packets, as in the previous set of experiments), which is expected to have a greater influence on the overall quality of service.

Figure 3.6a reports the instantaneous throughput (aggregated over the 50 subflows) for one of the 10 runs, collected every 0.5 s. This time, the drop in throughput when using *non-SR migration* is more critical, going to 0.2 Gbps. Figure 3.5b depicts the CDF of the instantaneous aggregate throughput as collected every 0.5 s, over the 10 runs. *Non-SR migration* still exhibits low performance, with the 5-th percentile for throughput being 0.6 Gbps, as compared to 1.9 Gbps for *SR migration*. Furthermore, in this case, there is a (small yet perceptible) benefit from buffering packets vs dropping them when using *SR migration*, as each x-th percentile for throughput is greater with *SR migration with buffer* than with *SR migration with drop*.

3.5 Summary of Results

This chapter has introduced a mechanism to perform zero-loss VM migration in IPv6 data-centers. Contrary to traditional approaches, which maintain connectivity to migrating VMs re-actively by updating the location of VMs after they have migrated, this chapter introduces a proactive mechanism, which pre-provisions a logical path through which packets will flow shortly before, during, and shortly after, migration.

This is enabled by the use of IPv6 Segment Routing, allowing to direct packets destined to a migrating VM through this logical path. This way, coupling between the data plane and the hypervisor is reduced to a minimum: the only operation that a virtual router performs upon receipt of a packet, is checking whether the corresponding interface is up, and forward the packet locally or to the next segment accordingly. Implementation on a real virtual router, VPP, and evaluation by way of diversified workloads, show that the proposed mechanism is indeed able to provide zero-loss VM migration, with benefits both in terms of TCP throughput and session opening latency.

Note that this chapter focuses on VMs processing traffic from/to the outside of the data-center, *i.e.,* egress-facing servers or virtual network functions. An interesting further question to be investigated is how to use similar techniques to provide seamless migration for VMs handling intra-data-center traffic.

Results from this chapter have been published in [79].

Chapter 4

Flow-Aware Workload Migration in Data Centers

With the desirability of task (VM or container, also occasionally called *"workload"* in literature [103]) mobility established in chapter 3, where it was also shown that it was possible to do so without losses, the question of an optimal task migration mechanism naturally arises, which will be investigated in this chapter. Task placement has been studied, and current approaches provide mathematical models focusing on optimal usage of resources and on flow minimization [93], or models that also consider migration but do not take the detailed network topology into account [104, 105].

This chapter introduces a multi-objective linear programming model of task migration that satisfies per-node resources constraints and inter-nodes communication requirements while minimizing the cost incurred by migrations. The fundamental assumptions are (i) that tasks have inter-communication demands that change with time, (ii) that tasks that communicate benefit from being topologically close, and therefore (iii) that changes in inter-communication demands may make a current task placement suboptimal, suggesting possible efficiency gains by relocation. Hence, the developed multi-objective linear programming model of task migration is intended to run iteratively, at regular time intervals. It uses task inter-communication demands to decide if and where to re-allocate tasks so as to provide the best possible task placement, while minimizing a migration cost. Network constraints are taken into account by modeling the traffic as a multi-commodity flow problem, where a commodity represents an inter-task communication demand.

4.1 Statement of Purpose

For some distributed computing applications, where tasks frequently communicate with each other, the limiting factor for the completion of a running task is not necessarily its actual location (assuming that the task is placed on a machine with the full physical resources it needs), but can be the throughput at which it exchanges data with other tasks. Furthermore, if those tasks have changing traffic requirements, migrating them within the data center can be an efficient way to re-organize the load across all available machines so as to optimally satisfy these requirements.

With these two assumptions as a baseline – tasks need to be placed according to their communication requirements, and those requirements can change over time – the multi-objective mathematical problem introduced in this chapter consists of determining a placement which satisfies machine, network and applications constraints, while (i) maximizing the number of new tasks to be assigned, (ii) minimizing the migration cost and (iii) maximizing the total inter-task throughput.

The contribution of this chapter is therefore threefold: (i) it formulates a model of task migration which uses an accurate description of network demands and of the network topology; (ii) it proposes a multi-objective approach which generates a set of Pareto-optimal solutions, from which the operator can choose according to operational tradeoffs; and (iii) it proposes a heuristic method that can generate an approximate Pareto front while greatly reducing solving time as compared to obtaining the exact solution.

Evaluations show that the sets of solutions generated by this approximation method remain close to their optimal counterparts, regardless of the topology and the task-to-machine ratio.

4.1.1 Related Work

Data center traffic has been extensively studied: it has been shown that the amount of intra-rack traffic is as voluminous as is extra-rack traffic [41], and that patterns emerge when considering the traffic matrix between interdependent tasks [106, 107]. Some task pairs communicate more than others, in which case it makes sense to locate those on either the same machine, or on machines close to each other in the network topology. Therefore, data center architectures have been studied which take communication dependencies between tasks into account, from a network systems perspective [42, 108]. This motivates the need for models that can place tasks according to their pairwise communication demands.

The task placement problem consists of finding an efficient assignment of tasks to machines within a data center, satisfying a set of constraints (resources, possibly traffic demand) with respect to one or more objective functions such as energy consumption minimization [91,103,109–111], communication cost minimization [92, 93], network traffic minimization or resource utilization maximization. Different optimization approaches have been proposed for the task placement problem: solution techniques include deterministic algorithms, heuristics, meta-heuristics and approximation algorithms. Surveys of these methods can be found in [90, 112–114]. Many approaches have been proposed that aim at optimizing a single objective. Multi-objectives formulations have also been introduced, but they are often reduced to a mono-objective problem, failing to provide a set of Pareto-optimal solutions.

Single-objective approaches: In [104], a model that considers resources constraints as well as dependencies between applications is proposed, which then re-allocates tasks while minimizing the migration impact. Contrary to the approach presented in this chapter, [104] does not consider changing communication demands between applications, but only migrate tasks located on overloaded machines. In [105], this model is refined by incorporating finer-grained server-side constraints, as well as network capacities of the machines. In [115], the problem of optimizing both task placement and traffic flow routing so as to save energy is considered. A model is formulated that minimizes the power consumption of switches and links, while satisfying capacity and traffic demands constraints. In [92], a network-aware model is considered that places tasks with the aim of minimizing communication cost. A greedy consolidation algorithm is proposed, consisting of identifying task clusters based on network traffic (using a cost matrix). In [107], a traffic-aware task placement model is introduced, where the objective is to minimize the aggregate traffic rates perceived by every switch. A two-tier approach given by a cluster-and-cut heuristic is proposed.

Multi-objective reduced to single-objective: Some VM placement problems are formulated as multi-objectives models, which are then reduced to single-objective models. In [116], the authors consider the placement of VMs on machines by minimizing the size of clusters of VM serving the same jobs, and minimizing the maximum traffic on uplinks of top-of-rack switches. An iterative least-loaded-first based placement algorithm is proposed, which first gives the priority to locality, and then to channel occupancy. In [117], a power-efficient VM placement and migration model is introduced, aiming at minimizing the number of machines used, the network energy consumption and the network end-to-end delay. The model is solved using a weighted sum approach. In [118], VM placement is considered under three objectives to be minimized: total resource "waste", power consumption, and thermal dissipation costs. The solution approach is a combination of a generic grouping algorithm and a fuzz multi-objective evaluation. In [119], a VM management framework considering initial VM placement and VM migration is introduced, with three objectives: elimination of thermal hotspots, minimization of power consumption and application performance satisfaction. The problem is solved using a weighted sum approach.

Pure multi-objective approaches: In [120], a bi-objective mathematical formulation for VM placement is proposed, aiming at minimizing resource wastage and power consumption. A multi-objective ant colony system algorithm is proposed to generate the set of non dominated solutions. In [121] (extending [122]), a multi-objective formulation of the VM placement problem is introduced, aiming at minimizing energy consumption and network traffic, and maximizing economical revenue while satisfying a service level agreement (SLA). A memetic algorithm given by an evolutionary process is proposed, and a Pareto set approximation is returned.

4.1.2 Chapter Outline

The remainder of this chapter is organized as follows. Section 4.2 introduces the framework needed to represent the state of the data center. Section 4.3 presents the mathematical formulation

Notation	Description
$i, i', \cdots \in I^t$	Tasks to be run at time t
$m, m', \cdots \in \mathcal{M}$	Machines
$r, r', \cdots \in \mathcal{R}$	Routers
$A \subseteq (\mathcal{M} \cup \mathcal{R})^2$	Network edges
$A_{mm'} \in 2^A$	Path $m \mapsto m'$
$c_{uv} > 0$	Capacity of link (u, v)
$\kappa_m > 0$	Machine CPU capacity
$\rho_i > 0$	Task CPU requirement
$\sigma_i > 0$	Task size
$C^t \subseteq I^t \times I^t$	Communicating tasks at time t
$d^t_{ii'} > 0$	Throughput demand for $i \mapsto i'$ at time t
$x^t_{im} \in \{0, 1\}$	Task i is placed on machine m at time t
$f^t_{ii'}(u, v) \in \mathbb{R}^+$	Flow for $i \mapsto i'$ along (u, v) at time t

Table 4.1 – Inputs and variables to the model

proposed for the flow-aware workload placement and migration problem, given by a multi-objective Mixed Integer Linear Program (MILP) [123]. Section 4.4 describes the algorithm used to generate the set of Pareto-optimal solutions to this problem, and provides a computational example. An approximation method is proposed in section 4.5; its performance in terms of solving time and quality of the generated approximate solutions (as compared to the optimal ones) are then evaluated. Finally, section 4.6 concludes this chapter.

4.2 Data Center Representation

Before detailing the problem formulation (section 4.3), resolution (section 4.4) and approximation (section 4.5), this section introduces the framework used to describe the state (including the topology) of a data center. This is achieved by considering (i) a set of machines, (ii) the network topology connecting them, and (iii) a set of tasks to place on these machines. As this model considers task migration, for which it is necessary to be aware of the evolution of the state of the data center, the model makes the assumption that it is running at a given time t (where t is an abstract index without unit), and that the previous state of the data center corresponds to time $(t-1)$. Thus as a convention, a variable superscripted with t represents the current state of the data center (to be optimized), and a variable superscripted with $(t-1)$ is a known input representing the previous state of the data center. A summary of the notations used throughout this chapter is provided in table 4.1.

4.2.1 Tasks and Machines

Tasks to be run at time t are represented by a set I^t. At time t, new tasks can arrive or existing tasks can finish executing. *Arriving tasks* correspond to those in $I^t \setminus I^{t-1}$. *Already existing tasks* correspond to those in $I^t \cap I^{t-1}$, from among which those that were successfully assigned to a machine at time $(t-1)$ have priority over new tasks and *must* be placed at some machine at time t (they can continue execution on the same machine, or be migrated to another, but cannot be stopped). *Terminated tasks* correspond to those in $I^{t-1} \setminus I^t$, and thus are not of interest to the model.

Machines are represented by a set $\mathcal{M}$. Each machine $m \in \mathcal{M}$ has a *CPU capacity* $\kappa_m > 0$ which represents the amount of work it can accommodate. Conversely, each task $i \in I^t$ has a *CPU requirement* $\rho_i > 0$, representing the amount of resources it needs in order to run. Finally, each task $i \in I^t$ has a *size*[1] $\sigma_i > 0$, which will be used to model the cost of migrating the task from one machine to another.

[1] For instance, the size of RAM plus storage for a virtual machine.

4.2.2 Network Model

In order to take application dependencies into account, the physical network topology existing between the machines must be known. To that end, and as described in section 1.4, a network is modeled as a directed graph $G = (V, A)$, where $V = \mathcal{M} \cup \mathcal{R}$ is the set of vertices, with $\mathcal{R}$ the set of routers and A the set of arcs. An arc $(u, v) \in A$ can exist between two routers or between a machine and a router, but also between a machine and itself to model a loopback interface (so that two tasks on the same machine can communicate). Each of those arcs represents a link in the network, and has a capacity of $c_{uv} > 0$. For each ordered pair of machines $(m, m') \in \mathcal{M}^2$, a list $A_{mm'} \in 2^A$ represents the path from m to m' (as illustrated in section 1.4).

Finally, at a given time t, an ordered pair of tasks $(i, i') \in I^t \times I^t$ can communicate with a *throughput demand* $d^t_{ii'} > 0$, representing the throughput at which i would like to send data to i'. Let $G^t_d = (I^t, C^t)$ be a weighted directed graph representing these communication demands, where each arc $(i, i') \in C^t$ is weighted by $d^t_{ii'}$. The graph G^t_d will be referred to as the *throughput demand graph*.

4.3 Mathematical Modeling

Based on the data center framework described in section 4.2, this section presents a multi-objective Mixed Integer Non Linear Program (MINLP) aiming at optimizing task placement and migration while satisfying inter-application network demands. A linearization as a multi-objective MILP is then derived, allowing for an easier resolution.

4.3.1 Variables

Two sets of variables are introduced, representing (i) task placement and (ii) the network flow for a particular placement.

Task placement The aim of the model is to provide a placement of each task $i \in I^t$ on a machine $m \in \mathcal{M}$, at a given timestep t. The binary variable x^t_{im} reflects this placement: $x^t_{im} = 1$ if i is placed on m, and $x^t_{im} = 0$ otherwise.

Network flow In order to determine the best throughput that can be achieved between each pair of communicating tasks, a variant of the *multi-commodity flow problem* [124, p. 58] is used, where a commodity is defined by the existence of $(i, i') \in C^t$. A commodity thus represents an inter-task communication.

For each link $(u, v) \in A$, $f^t_{ii'}(u, v)$ is a variable representing the throughput for communication from i to i' along the link (u, v).

4.3.2 Constraints

Two sets of constraints are used to model this flow-aware workload migration problem: *placement constraints* (equations (4.1-4.3)) represent assignment of tasks to machines, whereas *flow constraints* (equations (4.4-4.8)) focus on network flow computation.

The *placement constraints* represent the relationship between tasks and machines. First, each task $i \in I^t$ must be placed on at most one machine:

$$\sum_{m \in \mathcal{M}} x^t_{im} \leq 1, \forall i \in I^t \tag{4.1}$$

Forcefully terminating a task is not desirable: if a task was running at time $t-1$, and is still part of the set of tasks at time t, it must not be forcefully terminated:

$$\sum_{m \in \mathcal{M}} x^{t-1}_{im} \leq \sum_{m \in \mathcal{M}} x^t_{im}, \forall i \in I^{t-1} \cap I^t \tag{4.2}$$

where x^{t-1}_{im} is a known input given by the state of the system at time $t-1$. If the task was already successfully placed at time $t-1$ (*i.e.,* $i \in I^{t-1}$ and $\sum_{m \in \mathcal{M}} x^{t-1}_{im} = 1$) and is still to be run at time

t (*i.e.,* $i \in I^t$), the left-hand-side of the equation will be 1, thus forcing placement of the task at time t.

Finally, the tasks on a machine cannot use more CPU resources than the capacity of that machine:

$$\sum_{i \in I^t} \rho_i \, x^t_{im} \leq \kappa_m, \forall m \in \mathcal{M} \tag{4.3}$$

The *flow constraints* allow computing the throughput for each commodity (*i.e.,* ordered pair of communicating task). For each link $(u, v) \in A$ in the network, the total flow along the link must not exceed its capacity:

$$\sum_{(i,i') \in C^t} f^t_{ii'}(u, v) \leq c_{uv}, \forall (u, v) \in A \tag{4.4}$$

For a commodity $(i, i') \in C^t$, the flow going out of a machine m must not exceed the *throughput demand* for the communication from i to i'. Also, the flow must be zero if task i is not hosted by machine m:

$$\sum_{v:(m,v) \in A} f^t_{ii'}(m, v) \leq d^t_{ii'} \, x^t_{im}, \forall m \in \mathcal{M}, \forall (i, i') \in C^t \tag{4.5}$$

Conversely, for a commodity $(i, i') \in C^t$, the flow entering a machine m' must not exceed the *throughput demand* for the communication from i to i', and must be set to zero if task i' is not on m':

$$\sum_{v:(v,m') \in A} f^t_{ii'}(v, m') \leq d^t_{ii'} \, x^t_{i'm'}, \forall m' \in \mathcal{M}, \forall (i, i') \in C^t \tag{4.6}$$

Each router $r \in \mathcal{R}$ must forward the flow for each commodity – that is, the ingress flow must be equal to the egress flow:

$$\sum_{v:(u,v) \in A} f^t_{ii'}(u, v) = \sum_{v:(v,u) \in A} f^t_{ii'}(v, u), \forall u \in \mathcal{R}, \forall (i, i') \in C^t \tag{4.7}$$

Finally, if a task i is placed on machine m and a task i' on machine m', the corresponding flow must go through the path specified by $A_{mm'}$. Otherwise, the flow computed by the model could go through a non-optimal path or take multiple parallel paths – and for simplicity of interpretation, the latter is considered out-of-scope for this chapter. Hence, the flow needs to be set to zero for all edges that do not belong to the path from m to m':

$$\begin{aligned} f^t_{ii'}(u, v) \leq c_{uv}(1 - x^t_{im} \, x^t_{i'm'}), \\ \forall (i, i') \in C^t, \forall m, m' \in \mathcal{M}, \forall (u, v) \in A \setminus A_{mm'} \end{aligned} \tag{4.8}$$

This constraint has no side effect if task i is not on m or task i' is not on m', since in this case it reduces to $f^t_{ii'}(u, v) \leq c_{uv}$, already covered by equation (4.4).

4.3.3 Objective Functions

The migration model as presented in this chapter introduces three different objective functions, modeling (i) the placement of tasks, (ii) the overall throughput achieved in the network and (iii) the cost incurred by task migration, respectively. These functions depend on a placement, *i.e.,* on an assignment of all variables x^t_{im} and $f^t_{ii'}(u, v)$. Let $\mathbf{x}^t$ (respectively $\mathbf{f}^t$) be the characteristic vectors of the variables x^t_{im} (respectively $f^t_{ii'}(u, v)$).

The *placement objective* is simple and expresses that a maximal number of tasks should be successfully placed on a machine. When removing inter-task communication, this degenerates to a standard assignment problem wherein each task should be placed on a machine satisfying its CPU requirement, while also not exceeding the machine capacity. The placement objective function is simply the number of tasks which are successfully placed on a machine:

$$P(\mathbf{x}^t) = \sum_{i \in I^t} \sum_{m \in \mathcal{M}} x^t_{im} \tag{4.9}$$

The *throughput objective* expresses the need to satisfy applications *throughput demands*. Modeling network dependencies between tasks in a data center is usually (*e.g.,* in [104]) done through the use of a cost function depending on the network distance between two machines. Having introduced a representation of the physical network and of applications dependencies in section 4.2.2, it is possible to represent the overall throughput reached in the data center. To compute the throughput of the communication from a task i to a task i', it suffices to identify the machine m on which i is running and take the flow going out of this machine for this commodity: $\sum_{m\in\mathcal{M}} x^t_{im} \sum_{v:(m,v)\in A} f^t_{ii'}(m,v)$. This expression is quadratic in variables x^t_{im} and $f^t_{ii'}(u,v)$, but, owing to the fact that (4.5) constrains the flow to be zero for machines to which i is not assigned, can be simplified to $\sum_{m\in\mathcal{M}} \sum_{v:(m,v)\in A} f^t_{ii'}(m,v)$. Therefore, the overall throughput in the data center can be expressed as:

$$T(\mathbf{f}^t) = \sum_{(i,i')\in C^t} \sum_{m\in\mathcal{M}} \sum_{v:(m,v)\in A} f^t_{ii'}(m,v) \tag{4.10}$$

Since each machine has a loopback link in the graph G (*i.e.,* that $(m,m) \in A, \forall m \in M$), this formulation also covers the case where i and i' are hosted on the same machine.

Finally, the *migration cost* reflects the cost of moving tasks from one machine to another. The fundamental assumption is that tasks in a data center have communication demands that can evolve over time (modeled by $d^t_{ii'}$). This means that migrating a task to a machine topologically closer to those machines hosting other tasks with which it communicates, can be a simple way to achieve overall better performance. The migration cost incurred between two successive times $(t-1)$ and t is modeled as the sum of the sizes of tasks that have migrated. Since at time t, the assignment $\mathbf{x}^{t-1}$ is known, it is possible to know if a task $i \in I^t \cap I^{t-1}$ placed on a machine $m \in \mathcal{M}$ has moved, by comparing x^t_{im} to $(1 - x^{t-1}_{im})$. Also, if a task is to be shut off at time t (*i.e.,* $i \in I^{t-1} \setminus I^t$), it must not be part of the computation of the number of migrated tasks. The total migration cost from time $t-1$ to time t can thus be expressed as:

$$M(\mathbf{x}^t) = \sum_{i\in I^t\cap I^{t-1}} \sum_{m\in\mathcal{M}} \sigma_i \, x^{t-1}_{im}(1 - x^t_{im}) \tag{4.11}$$

Using these three objectives, it is possible to express the flow-aware placement and migration model as a multi-objective MILNP:

$$(\mathcal{P}) \begin{cases} \max T(\mathbf{f}^t) \\ \max P(\mathbf{x}^t) \\ \min M(\mathbf{x}^t) \\ \\ \text{subject to } \begin{cases} (4.1\text{-}4.8) \\ \mathbf{x}^t \in \{0,1\}^{I^t\times\mathcal{M}} \\ \mathbf{f}^t \in (\mathbb{R}^+)^{C^t\times A} \end{cases} \end{cases} \tag{4.12}$$

These objectives tend to compete with each other. If a task starts communicating with another task, migrating it to the same machine (or to a machine closer in the topology) can increase the throughput at which they communicate, thus increasing the *throughput objective* function – but this will incur an increase of the *migration cost*. Likewise, if a new task arrives at t and needs important CPU resources, placing it right away can consume CPU capacity that could have been used for co-locating tasks: increasing the *placement objective* can in some cases decrease the *throughput objective*.

As an illustration, consider two machines m_1 and m_2 with CPU capacities $\kappa_1 = 1$ and $\kappa_2 = 3$, with a bottleneck link (of capacity 1) in between. Assume that at $(t-1)$ there were two tasks i_1, i_2 with CPU requirements $\rho_{i_1} = 1$ and $\rho_{i_2} = 1$, and that i_1 was placed on m_1 and i_2 on m_2. At time t, i_1 starts to communicate with i_2 (with throughput demand $d^t_{i_1 i_2} = 2$), and a third task i_3 arrives with CPU requirement $\rho_{i_3} = 2$. Then, if this new task is to be placed on some machine, m_2 is the only machine with enough capacity to host it. But placing i_3 on m_2 would prevent a migration of i_1 from m_1 to m_2, which would have increased the throughput of communication $i_1 \mapsto i_2$. This situation is summarized in figure 4.1.

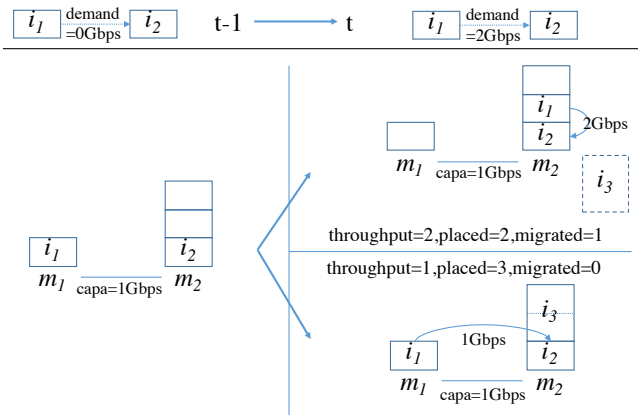


Figure 4.1 – Competition between objective functions

4.3.4 Linearization

All constraints expressed in section 4.3.2 are linear with respect to the variables x_{im}^t and $f_{ii'}^t(u,v)$, except for the *path enforcement* constraint in equation (4.8). Exploiting the fact that $x_{im}^t \in \{0,1\}$, this constraint can be linearized:

$$f_{ii'}^t(u,v) \leq c_{uv}(2 - x_{im}^t - x_{i'm'}^t), \\ \forall (i,i') \in C^t, \forall m,m' \in \mathcal{M}, \forall (u,v) \in A \setminus A_{mm'} \tag{4.13}$$

If $x_{im}^t \times x_{i'm'}^t \neq 1$, the equation will be $f_{ii'}^t(u,v) \leq c_{uv}$ or $f_{ii'}^t(u,v) \leq 2\ c_{uv}$, and will therefore be superseded by equation (4.4).

The set of constraints can be further compressed by writing only one equation per machine $m \in \mathcal{M}$ instead of one per tuple $m,m' \in \mathcal{M}$. This does not alter the model but makes the formulation more compact:

$$f_{ii'}^t(u,v) \leq c_{uv} \left(2 - x_{im}^t - \sum_{m' \in \mathcal{M}:(u,v) \notin A_{mm'}} x_{i'm'}^t\right), \\ \forall (i,i') \in C^t, \forall m \in \mathcal{M}, \forall (u,v) \in A \tag{4.14}$$

In this way, the flow-aware workload migration problem can be expressed as the following multi-objective Mixed Integer Liner Program (MILP):

$$(\mathcal{P}') \begin{cases} \max T(\mathbf{f}^t) \\ \max P(\mathbf{x}^t) \\ \min M(\mathbf{x}^t) \\ \text{subject to } \begin{cases} (4.1\text{-}4.7),\ (4.14) \\ \mathbf{x}^t \in \{0,1\}^{I^t \times \mathcal{M}} \\ \mathbf{f}^t \in (\mathbb{R}^+)^{C^t \times A} \end{cases} \end{cases} \tag{4.15}$$

4.4 Flow-Aware Workload Migration

This section describes an algorithm (Algorithm 3) solving the previously introduced multi-objective MILP (4.15). This algorithm runs at a given time t representing the "present". It takes the current inter-application communication requirements and the "past" (from time $t-1$) task placement as inputs, and returning a new placement as a solution.

4.4.1 Resolution Algorithm

A Pareto-optimal approach [123, p. 24] is used: a set of optimal solutions is generated, from which a final solution $\mathbf{z}^t = (\mathbf{x}^t, \mathbf{f}^t)$ can be extracted based on an operator-defined policy

Algorithm 3 Multi-Objective Migration Algorithm

$\mathbf{z}^{t-1} \leftarrow$ **input** ▷ *initial state*
$C^t, d^t_{ii'}, B, \underline{\varepsilon_p}, \overline{\varepsilon_p} \leftarrow$ **input** ▷ *parameters*
$\mathcal{S} \leftarrow \emptyset$
for $\varepsilon_p \in [\underline{\varepsilon_p}, \overline{\varepsilon_p}]$ with step δ_p **do**
  for $\varepsilon_m \in [B, 0]$ with step $-\delta_m$ **do**
    if $\exists \mathbf{z}' \in \mathcal{S}, P(\mathbf{z}') \geq \varepsilon_p \wedge M(\mathbf{z}') \leq \varepsilon_m$ **then**
      continue ▷ *dominant solution already existing*
    end if
    $\mathbf{z} \leftarrow$ solution of (4.18)
    ▷ *if* $\mathbf{z}$ *is not dominated, it is a candidate solution*
    if $\nexists \mathbf{z}' \in \mathcal{S}, \mathbf{z}' \succ \mathbf{z}$ **then**
      ▷ *remove any existing solution dominated by* $\mathbf{z}$
      for $\mathbf{z}' \in \mathcal{S}$ **do**
        if $\mathbf{z} \succ \mathbf{z}'$ **then**
          $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\mathbf{z}\}$
        end if
      end for
      $\mathcal{S} \leftarrow S \cup \{\mathbf{z}\}$ ▷ *add* $\mathbf{z}$ *to candidate solutions*
    end if
  end for
end for
▷ *select the best solution according to the considered policy*
$\mathbf{z}^t \leftarrow$ `getSolutionFromPolicy`($\mathcal{S}$)

`getSolutionFromPolicy()`. In the context of this migration model, a placement $\mathbf{z} = (\mathbf{x}, \mathbf{f})$ dominates a placement $\mathbf{z'} = (\mathbf{x}', \mathbf{f}')$ if all the objectives functions of the former have better values than the ones of the latter (with at least of one having a strictly better value). That is denoted $\mathbf{z} \succ \mathbf{z}'$:

$$\mathbf{z} \succ \mathbf{z}' \Leftrightarrow \begin{cases} P(\mathbf{z}) \geq P(\mathbf{z}') \\ T(\mathbf{z}) \geq T(\mathbf{z}') \\ M(\mathbf{z}) \leq M(\mathbf{z}') \\ (P(\mathbf{z}), T(\mathbf{z}), M(\mathbf{z})) \neq (P(\mathbf{z}'), T(\mathbf{z}'), M(\mathbf{z}')) \end{cases} \tag{4.16}$$

A set $\mathcal{S} = \{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$ of solutions is Pareto-optimal if and only if:

$$\forall (\mathbf{z}, \mathbf{z}') \in \mathcal{S}^2, \mathbf{z} \neq \mathbf{z}' \Rightarrow (\mathbf{z} \not\succ \mathbf{z}' \wedge \mathbf{z}' \not\succ \mathbf{z}) \tag{4.17}$$

To generate the set of Pareto-optimal solutions for the multi-objective problem, the ε-constraint method [123, p. 98] is used. It consists of reducing a multi-objective linear programming model to a single-objective model, by relaxing all objective functions but one and considering them as constraints. These constraints are obtained by bounding the relaxed objective function by a number ε that varies in a certain interval. Although its initial formulation only considers two-objectives problems, it can be extended to problems with a greater number of objective functions [125], as it is the case here.

The two functions that are relaxed are the *placement objective* and the *migration cost*, because it is simple to grid the corresponding space. The single-objective MILP that results is given by:

$$(\mathcal{P}_\varepsilon) \begin{cases} \max T(\mathbf{f}^t) \\ \text{subject to } \begin{cases} (4.1\text{-}4.7),\ (4.14) \\ P(\mathbf{x}^t) \geq \varepsilon_p \\ M(\mathbf{x}^t) \leq \varepsilon_m \\ \mathbf{x}^t \in \{0,1\}^{I^t \times \mathcal{M}} \\ \mathbf{f}^t \in (\mathbb{R}^+)^{C^t \times A} \end{cases} \end{cases} \tag{4.18}$$

where $\varepsilon_p, \varepsilon_m$ are bounds varying in $[\underline{\varepsilon_p}, \overline{\varepsilon_p}]$ and $[0, B]$ respectively.

For the *placement objective*, a simple upper bound for ε_p corresponds to the number of tasks:

$$\overline{\varepsilon_p} = \sum_{i \in I^t} 1 = |I^t| \tag{4.19}$$

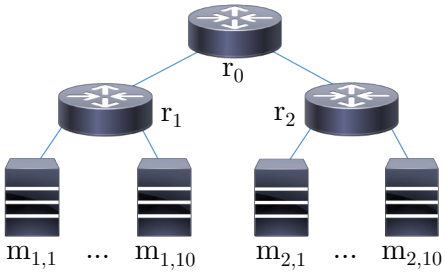


Figure 4.2 – Network topology used in the example run

$u = r_0, v \in \{r_1, r_2\}$	$c_{uv} = c_{vu} = 40$
$u \in \{r_1, r_2\}, v \in \mathcal{M}$	$c_{uv} = c_{vu} = 40$
$u \in \mathcal{M}$	$c_{uu} = 200$

Table 4.2 – Link capacities for the example topology

Similarly, a lower bound for ε_p can be easily derived from tasks that were running at the previous iteration and that are still to be run. As per equation (4.2), they cannot be forcefully terminated:

$$\underline{\varepsilon_p} = \sum_{i \in I^{t-1} \cap I^t} \sum_{m \in \mathcal{M}} x_{im}^{t-1} \tag{4.20}$$

For the *migration cost*, correctly choosing the upper bound B for ε_m has a large impact on the running time of the algorithm. This bound B will be referred to as the *migration budget*.

4.4.2 Computational Example

In large-scale deployments, the amount of CPU resources is likely sufficient for all tasks to be deployed. However, complex situations can exist, for which it is beneficial to have an understanding of the implications that different placement choices can have. For instance, when new tasks arrive that have a high CPU requirement, choosing to place them upfront can consume CPU capacity which could have been used to migrate a smaller task and satisfy its *throughput demand*. This section illustrates the model response in situations where placement and migration are competing with each other, by way of an environment in which the machines are nearly overloaded in the initial state. The algorithm has been implemented in Python using the Gurobi MILP solver [126], and all the simulations have been run on a virtual machine with 4 GB of RAM and an 2.6 GHz CPU.

Simulation Parameters

A simple two-tier tree topology is used, consisting of one root router r_0 linked to two ToR routers r_1, r_2, each linked to 10 machines (Figure 4.2). In order to represent the fact that tasks located on the same machine can achieve a higher throughput, each machine has a link to itself, with a higher capacity than other links in the data center, as depicted in Table 4.2.

The CPU capacity is set to 3 for each machine – which gives a total capacity of 60. An initial state with 40 tasks with CPU requirement $\rho_i = 1$ and a *throughput demand* graph G_d^{t-1} with 80 arcs of intensity $d_{ii'}^{t-1} = 10$ is considered, with an initial placement $\mathbf{z}^{t-1}$ that is optimal with respect to *throughput demands*[2].

Given this initial placement, this study explores the model behavior when communication demands of existing tasks change, while new bigger tasks arrive in the system. Communication demand changes are modeled by adding 10 random arcs (of intensity $d_{ii'}^t = 10$) between the already running tasks. Arrival of new tasks is modeled by creating 10 new tasks with CPU requirement $\rho_i = 2$, and by adding 10 random arcs (of intensity $d_{ii'}^t = 5$) going out of these tasks.

[2] $\mathbf{z}^{t-1}$ is obtained by starting from a random state and solving (4.15) where only the throughput objective is considered, with the additional constraint that all tasks must be allocated.

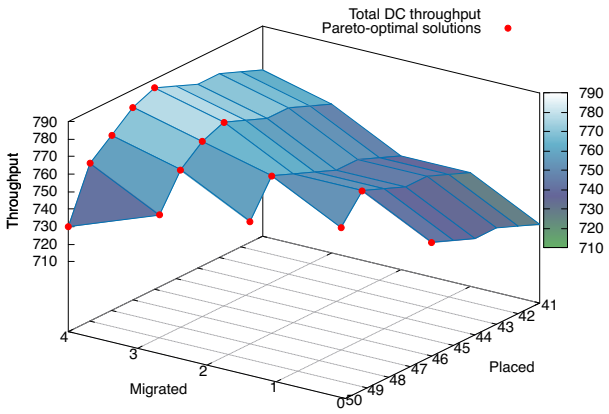


Figure 4.3 – Multi-objective model example run: solutions and Pareto-optimal solutions.

Results

In order to obtain the set of Pareto-optimal solutions $\mathcal{S}$ at time t, one iteration of Algorithm 3 was run. Figure 4.3 shows a three-dimensional representation of the solutions computed for one of those runs: Pareto-optimal solutions can be visually identified as those for which no other solutions have better coordinates in all dimensions. Results show that non-trivial behavior occurs when the data center is almost fully loaded: while placing a few new tasks improves the overall throughput, allocating all of them has a detrimental effect on the overall throughput because it prevents migration of other tasks.

This shows the interest of using a multi-objective model in cases where the objectives are conflicting with each other.

4.5 Pareto Front Approximation

Section 4.4 introduced a generic solution approach, which allows to compute the exact Pareto front for the three competing objectives (placing the highest number of tasks, minimizing the cost incurred by migration, achieving the best inter-application throughput) by way of an ε-constraint method. Although this approach generates an exact set of efficient solutions, and thus can be useful for small and highly-constrained deployments, its computational complexity makes it unsuitable to model larger infrastructures.

In order to overcome this issue, this section derives and experimentally analyses a heuristic for approximating the Pareto front. Section 4.5.1 introduces the approximation method, and section 4.5.2 provides the results of computational experiments in order to validate the quality of the proposed approach.

4.5.1 Heuristic Formulation

In large deployments, it is assumed that there are enough resources to accommodate all tasks to be run, thus the remainder of this section assumes that all tasks can be placed (*i.e.,* that there exists at least one solution to (4.18) such that $P(\mathbf{x}) = |I^t|$). In order to focus on migration, it is also assumed that no new task arrives (*i.e.,* $I^t = I^{t-1} := I$).

Given this assumption, a heuristic allowing to approximate the Pareto front in the *throughput-migration plane*, *i.e.,* for $(P(\mathbf{z}), T(\mathbf{z}), M(\mathbf{z})) \in \{|I|\} \times [0, +\infty) \times [0, B]$, is introduced. The issue with the ε-constraint approach introduced in section 4.4 is that, when considering the migration of B tasks, there are $\mathcal{O}(|I|^B |\mathcal{M}|^B)$ possible recombinations, which can be computationally expensive. In order to limit the combinatorics of the recombination, the introduced heuristic splits the migration process in smaller chunks, considering fewer migrations at a time, and proceeding in a greedy way until the total budget is exhausted, as described in Algorithm 4.

Algorithm 4 Heuristic Multi-Objective Resolution

$\mathbf{z}^{t-1} \leftarrow$ **input** ▷ *initial state*
$C^t, d^t_{ii'}, B \leftarrow$ **input** ▷ *parameters*
$\mu \leftarrow 0$ ▷ *total migration cost*
$\Delta \leftarrow$ **input** ▷ *incremental budget*
$\hat{\mathcal{S}} \leftarrow \emptyset$
$\hat{\mathbf{z}}^0 \leftarrow \mathbf{z}^{t-1}$
$\tau \leftarrow 1$ ▷ *step index*
while $\mu < B$ **do**
  ▷ *reduce Δ at last iteration so that $\mu + \Delta \leq B$*
  $\Delta \leftarrow \min\{\Delta, B - \mu\}$
  ▷ *find a placement within Δ of the previous one*

$$\hat{\mathbf{z}}^\tau \leftarrow \text{solution of} \begin{cases} \max T(\hat{\mathbf{f}}^\tau) \\ \text{subject to} \begin{cases} \text{(4.1-4.7), (4.14) with } t \mapsto \tau \\ P(\hat{\mathbf{x}}^\tau) = |I| \\ M(\hat{\mathbf{x}}^\tau, \hat{\mathbf{x}}^{\tau-1}) \leq \Delta \\ \hat{\mathbf{x}}^\tau \in \{0,1\}^{I \times \mathcal{M}} \\ \hat{\mathbf{f}}^\tau \in (\mathbb{R}^+)^{C^t \times A} \end{cases} \end{cases}$$

  if $M(\hat{\mathbf{x}}^\tau, \hat{\mathbf{x}}^{\tau-1}) = 0$ **then**
    break ▷ *stop if no task could be migrated*
  end if
  $\hat{\mathcal{S}} \leftarrow \hat{\mathcal{S}} \cup \{\hat{\mathbf{z}}^\tau\}$
  ▷ *compute the migration cost w.r.t. initial state*
  $\mu \leftarrow M(\hat{\mathbf{x}}^\tau, \hat{\mathbf{x}}^0)$
  $\tau \leftarrow \tau + 1$
end while
$\mathbf{z}^t \leftarrow$ `getSolutionFromPolicy`$(\hat{\mathcal{S}})$
return $\mathbf{z}^t$

For clarity, let $M(\mathbf{x}^t, \mathbf{x}^{t'})$ represent the migration cost when going from state $\mathbf{x}^t$ to $\mathbf{x}^{t'}$:

$$M(\mathbf{x}^t, \mathbf{x}^{t'}) = \sum_{i \in I \cap I^{t'}} \sum_{m \in \mathcal{M}} \sigma_i \, x^{t'}_{im}(1 - x^t_{im}) \tag{4.21}$$

Let $\Delta > 0$ be a parameter representing an *incremental budget*. Starting from an initial configuration $\mathbf{z}^{t-1}$, and given an upper-bound B on the migration cost, intermediary states ($\hat{\mathbf{z}}^{\mathbf{0}} = \mathbf{z}^{t-1}, \hat{\mathbf{z}}^{\mathbf{1}}, \ldots, \hat{\mathbf{z}}^\tau, \ldots, \hat{\mathbf{z}}^{\overline{\tau}}$) are created. At each step $\tau \geq 1$, a new placement is generated by maximizing the throughput objective, with the constraint that no more than Δ is incurred as a migration cost as compared to step $(\tau - 1)$. This way, no more than $\mathcal{O}(|I|^\Delta |\mathcal{M}|^\Delta)$ recombinations are considered at each step, allowing for a smaller expected computational complexity. This process ends when no further tasks can be migrated, or when a total migration cost of B (as compared to step 0) has incurred. The heuristic tracks the solutions generated while generating the successive solutions, so as to build an approximate Pareto front $\hat{\mathcal{S}}$.

4.5.2 Computational Experiments

This section experimentally illustrates the benefits of using the Pareto front approximation heuristic, both in terms of computational complexity and quality of the approximation. The sensitivity of the heuristic to the *incremental budget* parameter Δ is also experimentally explored. The experimental platform is the same as in section 4.4.2.

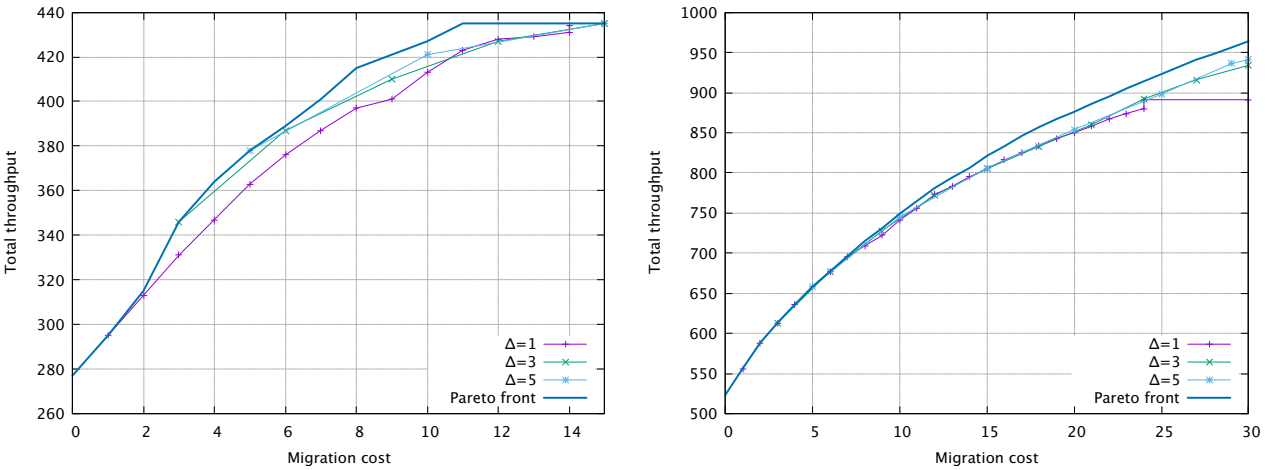
Simulation Parameters

In a common data center, all machines in a rack are linked to a top-of-rack (ToR) router, and those ToR routers are linked to each other via an aggregation router. In data centers, several such configurations can be linked together by way of a third routing layer – in all cases, resulting in a tree-like topology, as described in section 1.1.2.

For the purposes of these simulations, a *k-rack topology* is defined as a tree where a core router r_0 is attached to k ToR routers $r_1, r_2, \ldots, r_k$. Each ToR router r_h $(1 \leq h \leq k)$ is attached to

$u = r_0, v \in \{r_1, \ldots, r_k\}$	$c_{uv} = c_{vu} = 40$
$u \in \{r_1, \ldots, r_k\}, v \in \mathcal{M}$	$c_{uv} = c_{vu} = 40$
$u \in \mathcal{M}$	$c_{uu} = 1000$

Table 4.3 – Link capacities for the *k-rack topology*



(a) Tree topology, 20 machines, 40 tasks.

(b) Tree topology, 20 machines, 100 tasks.

Figure 4.4 – Pareto front approximation: Exact Pareto front and approximate Pareto fronts from Algorithm 4 for $\Delta \in \{1, 3, 5\}$.

10 machines $m_{h,1}, m_{h,2}, \ldots, m_{h,10}$. This topology thus comprises $10 \times k$ machines, and Figure 4.2 depicts a 2-*rack topology*. Links capacities are set as per Table 4.3, *i.e.,* links between the core router and the ToR routers, and between the ToR routers and the machines have a capacity of 40. Loopback interfaces on the machines have a capacity of 1000 – so as to favor placement of tasks which communicate with each other on the same machine.

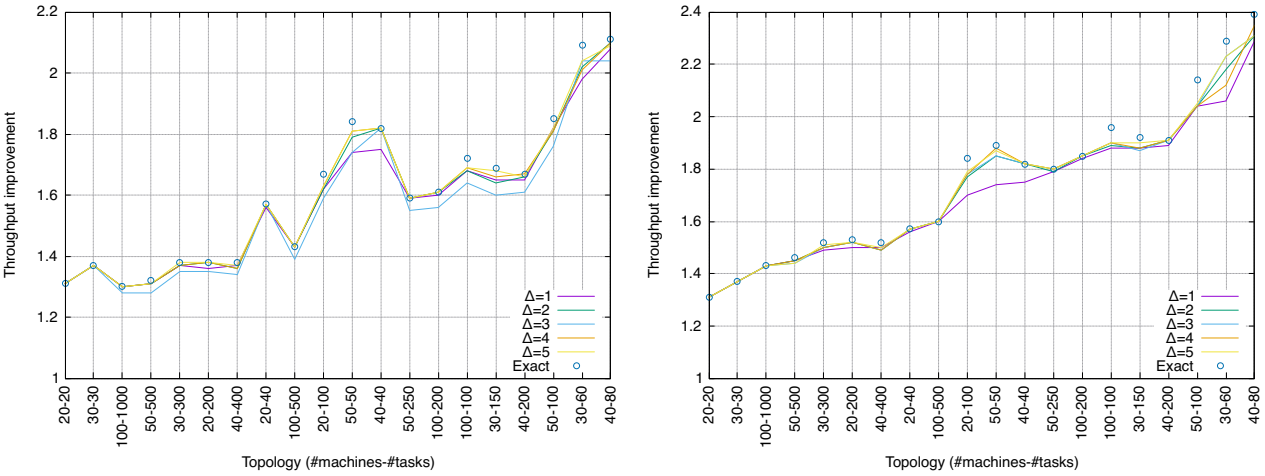
Five *k-rack topologies* are considered, for $k \in \{2, 3, 4, 5, 10\}$ (for a corresponding number of machines $|\mathcal{M}| \in \{20, 30, 40, 50, 100\}$). For each of these topologies, four scenarios are considered with different tasks-per-machine ratios, such that $|I| \in \{|\mathcal{M}|, 2|\mathcal{M}|, 5|\mathcal{M}|, 10|\mathcal{M}|\}$ – as a way to compare between different load factors for the machines. This makes for a total of $5 \times 4 = 20$ scenarios.

CPU requirements of tasks are set to $\rho_i = 1$, and CPU capacities of the machines to $\kappa_m = 20$. This allows for up to 20 tasks per machine, thus ensuring that all tasks can always be placed. To simplify understanding the migration cost, tasks size (for computing the migration cost) are set to $\sigma_i = 1$, thus the migration cost simply becomes the number of migrated tasks. Tasks are initially uniformly randomly assigned to the machines. The communication matrix is generated by: (i) for each task $i \in I$, uniformly choosing a random number D_i of destinations in $\{1, 2, 3\}$, (ii) for each of these D_i destinations, uniformly choosing a task $i' \in I \setminus \{i\}$ and (iii) setting a throughput demand $d_{ii'}$ for the communication $i \mapsto i'$ uniformly in $\{1, 2, \ldots, 10\}$.

For each of these 20 scenarios, the exact Pareto front is computed using Algorithm 3, with a migration budget (maximum migration cost) $B = 30$. The approximate Pareto front as described in Algorithm 4 is computed for five different values of the *incremental budget* $\Delta \in \{1, 2, 3, 4, 5\}$.

Comparative Results

In order to illustrate the results, the cases of the 2-*rack topology* (20 machines) with 40 and 100 tasks are first considered as examples. For these two cases, Figures 4.4a and 4.4b depict the generated exact and approximate Pareto fronts. Those figures show the different solutions from among which one can be chosen, after running either algorithm. As depicted in the figures, migrating more tasks obviously increases the migration cost – while enabling communicating tasks to be co-located, hence increasing the total achievable throughput.



(a) 20 migrations

(b) 30 migrations

Figure 4.5 – Pareto front approximation: throughput improvement for 20 and 30 migrations, θ_{20}/θ_0 and θ_{30}/θ_0. Exact Pareto front vs approximate Pareto fronts from Algorithm 4 for $\Delta \in \{1, 2, 3, 4, 5\}$. 20 different scenarios.

Depending on requirements, a solution can be chosen from the Pareto-optimal set that offers a tradeoff between improving the overall throughput and incurring more migrations. In these two examples, it can be observed that the approximate Pareto fronts generated by algorithm 4 for $\Delta \geq 3$ closely follow the exact front. It can also be observed that for $\Delta = 1$, results are of lower quality, and that in the example of Figure 4.4b, the algorithm was unable to find solutions for more than 24 migrations.

In order to quantify the impact of these algorithms on the quality of the obtained solution set, let θ_{20} (respectively θ_0, θ_{30}) represent the throughput corresponding to a migration cost of 20 (respectively 0, 30) in the Pareto front. More formally, this means that θ_{20} is the coordinate such that $(|I|, \theta_{20}, 20) \in \mathcal{S}$; for instance, in Figure 4.4b, $\theta_{20} = 876$ for the exact Pareto front. Figure 4.5a (respectively 4.5b) shows the *throughput improvement* θ_{20}/θ_0 (respectively θ_{30}/θ_0) for the 20 tested scenarios, for the exact Pareto front as well as the approximate Pareto fronts. This represents the relative improvement in terms of overall throughput if the operator chooses to migrate 20 (respectively 30) tasks after inspecting the set of Pareto-optimal solutions. It appears that, for $\Delta \geq 2$, the achievable throughput improvement when using the heuristic method introduced in section 4.5.1 is close to the exact solution. Overall, choosing $\Delta = 5$ yields the better results: for 20 migrations, the results are better in all the twenty tested instances, and for 30 migrations they are outperformed by $\Delta = 4$ in only three instances. For $\Delta = 5$, the throughput improvement remains within a 3% error as compared to the exact solution after 20 migrations, and within a 5% error after 30 migrations. Using $\Delta = 1$ yields results of lower quality, because the algorithm is not able to explore reconfigurations with two migrations at the same time (*i.e.,* swapping tasks), and because it can terminate prematurely if the throughput cannot be improved by migrating only one task. For 20 migrations, $\Delta = 3$ also yields results of lower quality, but this is more of an artificial nature: the number of migrations considered is 18 instead of 20 (since 20 is not a multiple of 3).

Finally, Figure 4.6 depicts the time needed to generate the exact and approximate Pareto fronts. While computing the exact Pareto front is very time-consuming, the approximation heuristics introduced reduces the required computation time. Overall, the lowest computation times are achieved for $\Delta \in \{3, 4, 5\}$, whereas $\Delta \in \{1, 2\}$ exhibit higher running times. The choice $\Delta = 5$, which can be observed to exhibit the best results, in terms of throughput improvement, in the majority of cases – with a computation time constantly better than the reference exact approach, performing approximately one order of magnitude faster when $B = 30$ (from 5.1× faster for 100 machines and 200 tasks, to 1800× faster for 30 machines and 60 tasks).

To conclude, the approximation method presented in section 4.5.1 can greatly reduce the time needed to compute a set of approximate Pareto optimal solutions, while generating solutions close enough to the exact ones. It is realistic to note that, although an improvement in solving time is observed regardless of the topology size, Figure 4.6 suggests that using a MILP approach (exact

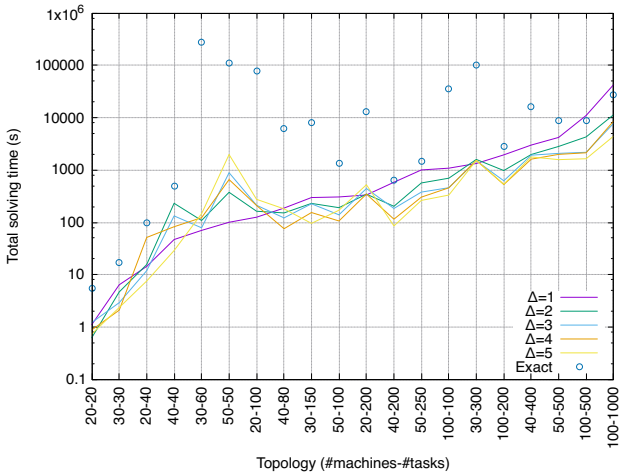


Figure 4.6 – Pareto front approximation: time to generate the approximate Pareto front, for up to $B = 30$ migrations. Exact Pareto front (Algorithm 3) vs approximate Pareto fronts (Algorithm 4) for $\Delta \in \{1, 2, 3, 4, 5\}$. 20 different scenarios.

or approximate) might be unfit for too large topologies.

4.6 Summary of Results

In data centers, applications may exhibit time-varying communications dependencies, and migrating tasks in such environments can be a means to re-distribute the load over the whole infrastructure, while optimizing inter-task communications. In this chapter, a linear programming framework is introduced, which represents the data center topology, and communications dependencies between tasks. Using this framework, a multi-objective model is developed, which, when solved, can propose a reorganization of tasks so as to (i) place the highest number of new tasks, (ii) maximize the overall achievable throughput, and (iii) minimize the migration cost incurred by this reorganization.

The approach developed is generic with respect to the network topology and the inter-application communication demands, and is novel in that, when compared to other workload migration frameworks, it generates a set of Pareto-optimal solutions. From among this solution set, a placement can be chosen, which yields the best tradeoff in terms of throughput improvement versus the cost incurred by task migration. A solution algorithm using the ε-constraint method is proposed as a baseline. Then, a heuristic is presented, which generates an approximate Pareto front. Computational experiments on different topologies show that the proposed approximation builds solutions close to the optimal Pareto front, while greatly reducing the computation time.

Having shown the desirability of considering inter-application dependencies in migration models, an interesting question that arises is how the proposed model can be extended to take into account more fine-grained parameters. These could include the distance between source and destination hosts when migrating, the energy cost induced by machines which are on or off, and the possibility to load-balance flows across multiple paths.

Results from this chapter have been published in [80].

Part III

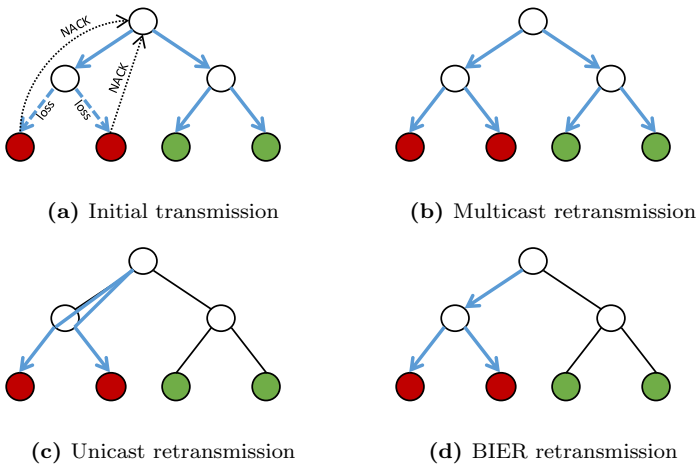
Reliable Content Distribution

Chapter 5

Reliable Multicast with BIER

Developed alongside their unicast counterparts, multicast protocols were never offered as universally available network services in the Internet [67] – in part, as the operational complexity of multicast was perceived as exceeding the potential benefits from efficient one-to-many distribution of content. "Native" multicast was therefore, when available, confined to within single networks, or even to within single links – and multicast between sites ("inter-network multicast") was established by way of overlays (*e.g.,* MBONE [127]). The complexity of multicast protocols is in part due to their group-based nature: schematically, when a client wishes to receive messages sent to a multicast group, it will explicitly and periodically send *join* messages to its "local multicast router" (using IGMP). This router will forward these *join* message upwards in the multicast tree (using *e.g.,* PIM [66]), until reaching the multicast source. Intermediate routers are expected to build, and maintain, flow state (*a minima*, a multicast tree) for as long as *join* messages are regularly received, in order to provide connectivity to all members of the multicast group.

Bit-Indexed Explicit Replication (BIER) [8] was designed to eliminate this complexity, and to enable lightweight inter-network multicast – with the ambition being that intermediate routers maintain no flow state, other than that of an existing unicast routing table, and that intermediate routers are not involved in group management. The key idea in BIER, as described in section 1.2.2, is that the source of a multicast data packet encodes the set of destinations (*i.e.,* the members of the group) as a bitstring, and includes this bitstring in the header of each multicast data packet. Intermediate routers only need to be able to interpret that bitstring – leaving group management a matter for only the clients and the source.



(a) Initial transmission **(b)** Multicast retransmission

(c) Unicast retransmission **(d)** BIER retransmission

Figure 5.1 – Comparison of different reliable multicast mechanisms. In this example, two clients do not receive a packet and send a NACK to the source (a). Multicast retransmissions (b) will vainly incur traffic towards the clients that had successfully received the packet. With unicast retransmissions (c), identical packets will be transmitted on the same link. BIER retransmissions (d) ensure that the traffic footprint is minimal.

5.1 Statement of Purpose

The original BIER specification [8] emphasizes the flexibility provided by BIER at the flow level, allowing for adding to and removing from the set of destinations of a flow without the need of building multicast trees and maintaining flow state in the routers. It is possible to take this one step further to modifying the set of destinations on a per-packet basis, and to do so without incurring any additional overhead.

This flexibility can be used to develop efficient reliable multicast: if the source of a multicast data packet is informed as to the set of destinations to which a retransmission is required, it can use BIER for minimizing the traffic footprint of this retransmission: by setting the bitstring so as to contain only the destinations affected by a multicast data packet loss, the retransmission will be forwarded only along the shortest path tree covering the source and these destinations.

This chapter studies this use of BIER for reliable multicast – and compares this "*reliable BIER*" to two reliable multicast references: (i) the mechanisms known from *e.g.,* NORM [128], in which the source, when informed about a destination in the multicast group being affected by a multicast data packet loss, will retransmit the multicast data packet to all destinations, and (ii) retransmission by way of unicast(s) to those destinations affected by a multicast data packet loss: this is for instance the case in RMTP [129], if the number of affected destinations is below a given threshold.

Figure 5.1 illustrates the intuition that when faced with a multicast data packet loss (figure 5.1a), a NORM-style retransmission from the source and to the entire multicast group (naturally) will impose a load on all links in the multicast tree, regardless of if they lead to destinations affected by a multicast data packet loss (figure 5.1b). Unicast retransmissions (figure 5.1c), while traversing only the shortest-path tree between the source and the destinations affected by a multicast data packet loss, may cause the same multicast data packet to be retransmitted across the same link multiple times – whereas BIER utilizes only the shortest path tree between the source and the destinations affected by a multicast data packet loss, with each packet only retransmitted across the same link once (figure 5.1d). This chapter formalizes a simple, reliable, multicast mechanism using BIER, and examines if the suggested intuition holds – and in which conditions. To that purpose, network simulations are conducted, and an analytical model is developed.

(Semi-)Reliable Multicast

While the term "reliable" is used throughout this chapter, and is generally used in the literature, it is perhaps more realistic to describe the attained multicast network services as semi-reliable. For example, maintaining a retransmission buffer (regardless if centralized at the source or distributed/peer-based) indefinitely is hardly feasible – nor will excessive retransmissions necessarily increase the overall success rate across heavily congested paths. Therefore, this chapter will not consider mechanisms whereby a source adapts its sending rate to the worst destination; rather, it will assume that the source sends a stream at a fixed rate (*e.g.,* a live broadcast media stream), and that destination applications might decide to give up on certain packets – if they are behind heavily congested links, and/or if they no longer would need the packet after retransmission.

5.1.1 Related Work

While never widely deployed as an inter-networking service, several reliable multicast protocols have been developed [130]. Reliable multicast protocols assume the existence of a multicast tree, to which are added (i) detection, (ii) reporting, and finally (iii) repairing of packet losses – the latter, through successful retransmission of lost packets. These mechanisms differ, depending on how *reliability* is understood, on the intended set of receivers, on transmission requirements, and on which trade-offs are acceptable for a given usage [131].

Loss estimation and recovery can be handled exclusively at the source, such as with the "xPress Transfer Protocol" (XTP) [132], a sender-reliable protocol (using acknowledgements and retransmission timers) designed for small sets of receivers. For larger multicast groups, the "Reliable Multicast Protocol" (RMP) [133] pushes responsibility for detecting losses and requesting retransmissions to the receivers, through NACKs. "Log-Based Receiver-reliable Multicast" (LBRM) [134] uses a *log server* for caching packets sent by the source, which also reacts to requests for retransmissions. A hierarchical architecture is suggested: a destination that has not received a packet will first solicit retransmission from a local *log server* – and only if that fails, will solicit a primary *log*

server. With the "Reliable Multicast Transport Protocol" (RMTP), [129] proposes a hierarchical architecture, wherein intermediate routers in the multicast tree will contribute retransmissions in case of isolated losses, but with global recovery handled by the source. [135] introduces "Scalable Reliable Multicast" (SRM), which uses receiver-based reliability (receivers detect losses and request retransmissions) combined with low-rate multicast by every member to report the highest sequence number received. The "Tree-based Multicast Transport Protocol" (TMTP) [136] is another instance of a reliable multicast protocol, wherein destinations are grouped in a tree of different domains, within which local recovery can be performed.

The IETF standardized "Negative-acknowledgment Oriented Reliable Multicast" (NORM) [137], using NACKs and source-based retransmissions to attain reliability. NORM also proposes redundancy and recovery by way of Forward Error Coding (FEC) – either proactively, or in response to negative acknowledgements. A TCP-friendly congestion-control mechanism has been proposed in [138]. The IETF has also examined the "Pragmatic General Multicast" (PGM) protocol [139], a hierarchical receiver reliable protocol, targeting large multicast groups. Losses are reported via unicast NACKs and recovery can be performed by retransmission from a *Designated Local Repairers* (*i.e.,* not necessarily from the source), also via multicast. This is similar to RMTP, in which retransmissions are performed by *Designated Receivers* – though RMTP supports both unicast and multicast retransmissions, depending on the number of reported losses of a given packet.

In the context of data-centers, [140] proposes the end-host based protocol "Reliable Data Center Multicast" (RDCM): through a central controller, RDCM explicitly builds a multicast tree, and a multicast-tree-aware backup overlay, for data dissemination. Retransmissions are performed on a peer-to-peer (unicast) basis: every receiver is responsible for providing, if needed, retransmissions for up to two of its peers.

The performance of reliable multicast protocols has been studied both analytically and through network simulations. For example, [141] develops an analytical model and carries simulations to study the performance of a generic reliable block-based multicast protocol using stop-and-wait, positive acknowledgements, and selective retransmissions. This model quantifies the number of transmission attempts until full reception, assuming independent losses in different links. [142] investigates the optimal placement of FEC in reliable multicast trees, by way of studying generic models of such trees (*i.e.,* a single path common to all receivers, a set of completely separate paths to each receiver) and a refinement of the model developed in [141]. The number of successful receptions for different types of trees is studied by way of analysis and simulation in [143], which also derives a generic approximation for the expected number of transmissions for reliable delivery.

Finally, models relying on TCP overlays for multicast have also been developed: [144] introduces the *One-to-Many TCP Overlay* for reliable multicast services, as an application-level multicast alternative to IP reliable multicast; [145] studies the performance of TCP-based reliable multicast trees, built as a set of reliable point-to-point links, in data-centers.

5.1.2 Chapter Outline

The remainder of this chapter is organized as follows: section 5.2 details the use of BIER for light-weight, reliable multicast. Section 5.3 and section 5.4 evaluate the performance of this reliable multicast mechanism (denoted *reliable BIER*) in different topologies and with different losses, by way of network simulations – and compare the performance with other reliable multicast mechanisms. Generalizing the observations from the network simulations, section 5.5 provides an analytical study of *reliable BIER* performance. Finally, section 5.6 concludes this chapter.

5.2 Reliable BIER – Specification

To provide reliable multicast as a network service transparent to applications, *reliable BIER* is designed to operate as a shim-layer above the network layer, as depicted in figure 5.2. This *reliable BIER shim layer* is upper-layer agnostic, and as such supports both standard transport layers (such as UDP) and encapsulation mechanisms such as IP-in-IP.

The *reliable BIER shim layer* assumes a unique flow ID from the upper layer, and maintains for each flow ID a sequence number, monotonically increased for each new packet being handed by the upper layer. The tuple (flow ID, sequence number) allows uniquely identifying each original multicast data packet in the network, identifying when a multicast data packet is received out of order, *etc.* For many transport layers, the flow ID would be a hash of the tuple (protocol number,

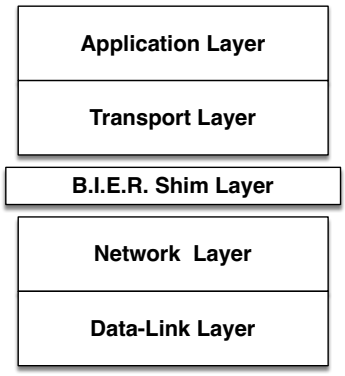


Figure 5.2 – BIER shim layer

source IP address, source port), whereas for more specific transport layers a transport-layer-specific identifier could be used (*e.g.,* the session ID for QUIC [146]).

For each outgoing multicast data packet, the *reliable BIER shim layer* takes a destination set, rather than a single destination IP address, from the upper layer – and compresses this into the destination-bitstring. The destination-bitstring and the sequence number comprise the *reliable BIER header*, to be included in each multicast data packet. The precise form of the *reliable BIER header* has no algorithmic significance – but in an IPv6 context can, for instance, take the form of a destination options extension header.

For each incoming multicast data packet, the *reliable BIER shim layer* at each destination will inspect the sequence number to detect losses, and signal losses by way of sending negative-acknowledgements (NACKs) from destinations towards the source. Having this signaling only involve the end-points (source and destinations) serves, in part to run the protocol over any "standard" set of BIER routers doing best-effort forwarding[1], and in part to facilitate deployment (only the end-points need to agree on parameters, for example). As in other reliable multicast protocols, *e.g.,* [128, 139], NACKs are used in order to avoid an "ACK storm". Of course, for very large error rates or errors affecting a wide range of destinations, this may lead to a "NACK storm". Such a storm of control traffic could be avoided *e.g.,* by allowing intermediate routers to aggregate NACKs before forwarding them upwards, without changing the end-to-end behavior specified in this chapter.

For the purpose of this chapter, a slightly modified socket API is used – specifically, allowing the sender to provide the set of destinations (rather than a single multicast group address) to which a multicast data packet is to be forwarded.

5.2.1 Source Operation

A *reliable BIER* source operates as detailed in algorithm 5. On sending a packet through a socket, the *reliable BIER shim layer* caches a copy of the packet, with which it associates a *BIER retransmit bitstring* with all bits cleared, and a timer of duration Δt_{agg}, where Δt_{agg} represents a window of time between the first NACK is received and a retransmission is made. Within this window, for each NACK received, the corresponding bit in the *BIER retransmit bitstring* is set; at the end of this window, the cached multicast data packet is retransmitted with the destination-bitstring set to the associated *BIER retransmit bitstring*. This permits aggregation of retransmissions to multiple destinations in a single BIER packet, thus potentially reducing the number of transmissions of this packet. If after retransmission, a subsequent NACK for the same packet is received, a new Δt_{agg} window is opened and the aggregation mechanism is restarted.

[1] It is to be noted, however, that if intermediate routers provides caching capabilities (as for instance in Information Centric Networks [22]), they could be extended to intercept NACKs and perform retransmissions in place of the source. This would not change the end-to-end behavior as specified in this chapter.

Algorithm 5 *Reliable BIER* Source Operation

$\Delta t_{agg} \leftarrow$ NACK aggregation delay
$w \leftarrow$ packet cache window size
$F \leftarrow$ unique flow identifier
$B \leftarrow$ destination bitstring
$S \leftarrow 0$ ▷ *sequence number*
$C \leftarrow \{\}$ ▷ *packet cache*
$R \leftarrow \{\}$ ▷ *retransmit bitstrings*
$T \leftarrow \{\}$ ▷ *retransmit timers*
for each outgoing packet p **do**
  insert *reliable BIER* header with flow F, seq S
  insert BIER header with bitstring B
  transmit p
  $C[S] \leftarrow p, R[S] \leftarrow 0, T[S] \leftarrow \infty$
  delete $C[S-w], R[S-w], T[S-w]$ ▷ *garbage collection*
  $S \leftarrow S + 1$
  for $s \in C$ **with** $T[s] \leq T_{now}$ **do** ▷ *perform retransmits*
    $p \leftarrow C[s]$ ▷ *retrieve cached packet*
    insert *reliable BIER* header with flow F, seq s
    insert BIER header with bistring $R[s]$
    transmit p
    $R[s] \leftarrow 0, T[s] \leftarrow \infty$
  end for
end for
for each received NACK packet with flow F, seq s, bit b **do**
  if $s \in C$ **then**
    $R[s] = R[s]$ OR 2^b ▷ *add b to the retransmit bitstring*
    $T[s] \leftarrow \min\{T[s], T_{now} + \Delta t_{agg}\}$ ▷ *schedule retransmit*
  end if
end for

5.2.2 Destination Operation

A *reliable BIER* destination operates as described in algorithm 6. In short, a destination will send a NACK to the source when it detects that a packet was lost – a packet being deemed lost when one of its successors is received[2]. If necessary, NACKs for a lost packet are then retransmitted regularly by way of a timer, until a retransmission is received.

More precisely, for each incoming multicast data packet, the *reliable BIER shim layer* parses the *reliable BIER header* and either hands it off to the upper layer, or (if received out-of-order) records it in a buffer, C. For each multicast data packet that (1) is received out-of-order, and (2) creates a "hole" (*i.e.,* a set of missing packets between two consecutively received packets) in C, the *reliable BIER shim layer* adds the element(s) corresponding to this "hole" in the list of lost packets, L. Each element in L is identified by the sequence number s of the corresponding lost packet, and is associated with a timer $T[s]$, and a NACK count $N[s]$.

For each element of L, a NACK is sent towards the source. The NACK contains a *reliable BIER header* wherein the included bitstring indicates the bit of the client sending the NACK[3], and the sequence number corresponding to the lost packet. Then, the NACK count is incremented, and a new timer for this packet is set to expire after Δt_{retry} (a configurable retry delay). Upon timer expiration, if no retransmission has been received, and if the NACK count is below a configurable limit ℓ, another NACK is sent and the process is restarted. When the retransmission count reaches ℓ, it is assumed that the source is not able to offer timely retransmission (for instance, due to congestion on the path), and the destination gives up trying to request. This achieves a *"poor-man's congestion control"*, by limiting the number of possible retransmission of a multicast data packet.

[2]The successor of the last data packet is a special end-of-connection packet. To ease readability, Algorithms 5 and 6 assume an infinite stream.

[3]One reason for including a bitstring is, that this allows the originator to create the and *BIER retransmit bitstring* by a simple OR operation of received NACKs.

Algorithm 6 *Reliable BIER* Destination Operation

$\Delta t_{retry} \leftarrow$ NACK retransmission delay
$\ell \leftarrow$ NACK retransmission limit
$F \leftarrow$ unique flow identifier
$L \leftarrow \{\}$ ▷ *seqnum of lost packets*
$T \leftarrow \{\}$ ▷ *NACK transmit timers for lost packets*
$N \leftarrow \{\}$ ▷ *number of times a NACK has been sent*
$C \leftarrow \{\}$ ▷ *recovered packets pending for app*
$n \leftarrow 0$ ▷ *next expected seqnum*
for each incoming packet p with flow F, seq S **do**
  if $S = n$ **then** ▷ *received in-order packet*
    $L \leftarrow L \setminus \{S\}$, delete $T[S], N[S]$ ▷ *un-schedule NACK*
    transmit p to application
    $n \leftarrow n + 1$
  else if $S > n$ **then** ▷ *received out-of-order packet*
    if $S \notin C$ **then** ▷ *cache packet and un-schedule NACK*
      $C[S] \leftarrow p$
      $L \leftarrow L \setminus \{S\}$, delete $T[S], N[S]$
    end if
    ▷ *schedule NACK for packets between n and S*
    for *seq* **from** n **to** S **with** *seq* $\notin L \cup C$ **do**
      $L \leftarrow L \cup \{seq\}, T[seq] \leftarrow T_{now}, N[seq] \leftarrow 0$
    end for
  end if
  ▷ *send appropriate NACKs*
  for *seq* $\in L$ **with** $T[seq] \leq T_{now}$ **do**
    if $N[seq] < \ell$ **then**
      send NACK with flow F, seq *seq*
      $T[seq] \leftarrow T_{now} + \Delta t_{retry}, N[seq] \leftarrow N[seq] + 1$
    else ▷ *abort trying to recover this packet*
      $L \leftarrow L \setminus \{seq\}$, delete $T[seq], N[seq]$
      $C[seq] \leftarrow \{\}$ ▷ *put a dummy packet in the cache*
    end if
  end for
  ▷ *send pending recovered packets to application*
  while $n \in C$ **do**
    $p \leftarrow C[n]$, delete $C[n]$
    transmit p to application
    $n \leftarrow n + 1$
  end while
end for

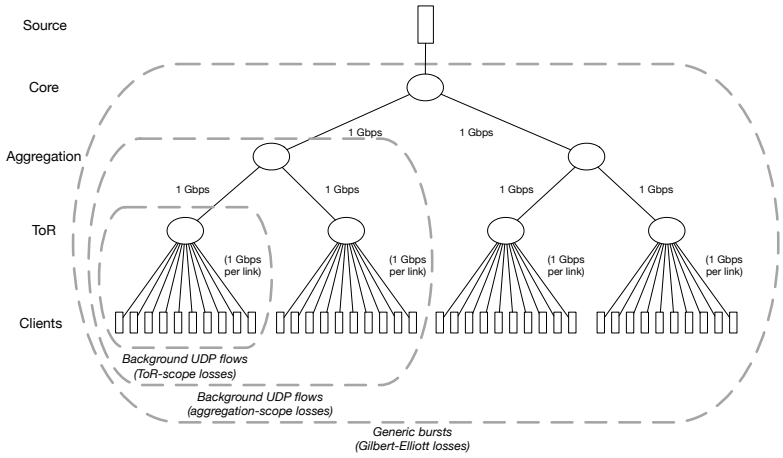


Figure 5.3 – Data-center simulation topology

5.2.3 Parameter Discussion

The value of Δt_{agg} directly influences the global behavior of *reliable BIER*:

- when $\Delta t_{agg} = 0$, no aggregation is performed, and the protocol degenerates to individual, unicast-based, retransmissions;
- when $\Delta t_{agg} > 0$, aggregation is enabled: the greater Δt_{agg}, the greater the probability of aggregating retransmissions, but at the cost of a greater delay for recovery;
- when $\Delta t_{agg} = RTT_{max} - RTT_{min}$, maximum aggregation is enabled; higher values of Δt_{agg} will not provide further benefit.

Determination of the value of Δt_{agg} thus requires taking into account RTT variations and error probability along the different paths, as well as the sensitivity of the application to delay: it thus corresponds to a policy decision. Furthermore, to make sure that each destination sends at most one NACK for each multicast data packet (re)transmission failure, Δt_{retry} should be greater than $\Delta t_{agg} + RTT_{max}$.

5.3 Data-Center Simulations

Regardless of the underlying network topology, content delivery with BIER from a given source will follow (but not construct) a shortest-path tree. Thus for this first set of simulations, *reliable BIER* is tested on a simple tree-topology modeling a data-center, depicted in figure 5.3: a core router, connected to two aggregation routers – each of which is connected to two Top-of-Rack routers (ToR), and with each rack hosting 10 machines.

The purpose of the set of tests in this section is to examine if the intuition introduced in section 5.1, and depicted in figure 5.1, holds: that using BIER (rather than multicast or unicast) for retransmissions can yield a measurable and significant diminution of the traffic footprint.

To this end, three different scenarios are constructed around the same physical topology depicted in figure 5.3. These scenarios serve to explore how *reliable BIER* performs both when losses are spatially located and when they are not, specifically:

Uncorrelated localized losses, where background traffic is present inside the leftmost rack, *i.e.,* where both the source and destination of the background traffic are members of the leftmost rack, saturating individual links between the ToR switch and the machines in the rack, and thus affecting these machines individually – but with the rest of the data-center unaffected. This scenario is studied in section 5.3.2.

Correlated localized losses, where background traffic is present inside the two leftmost racks, *i.e.,* where source and destination are members of the two leftmost racks. This saturates the incoming links to the two leftmost ToR switches, and thus will affect destinations on all

machines within a rack together – again, with the rest of the data-center unaffected. This scenario is studied in section 5.3.3.

Bursty, non-localized losses, where losses are not related to localized background traffic, but are produced by a loss model in each individual link. This scenario is studied in section 5.3.4.

5.3.1 Simulation Parameters and Setup

The BIER shim layer described in section 5.2 has been implemented in `ns3` [147], with the bit number $\mapsto$ IP-address mapping assumed *a priori* available in all routers, as discussed in section 1.2.2. UDP is used as transport protocol, and BIER and *reliable BIER* headers are implemented as IPv6 extension headers. The *reliable BIER* parameters from section 5.2 are chosen as follows: $\Delta t_{agg} = 7$ ms (NACK aggregation delay), $\Delta t_{retry} = 15$ ms (NACK retransmission timer), and $\ell = 3$ (retransmission limit). All links are homogeneous, point-to-point, with 1 Gbps capacity, with an MTU of 1500 octets and a propagation delay of 1 μs. Network interfaces all have tail-drop queues of size 512 packets. The links themselves are lossless – except for the simulations in section 5.3.4, which considers link-losses according to a Gilbert-Elliot loss model [148]. For these simulations, the multicast data packet source is attached to the core router, which generates a constant bit-rate *reliable BIER* flow of 500 Mbps. All 40 machines are destinations for this flow.

This scenario can be considered to represent *e.g.,* broadcasting of a live media, where a constant transmission bitrate has to be sustained, and where a retransmitted multicast data packet is of no value if received "too late". Thus, some packets may not be received by all destinations, and the ratio of packets successfully received after retransmissions (the *delivery ratio*) will be a metric of interest – as will the network load of the different links in the network, as well as the sum of traffic over all links in the network (the *traffic footprint*).

When unicast background traffic is introduced in the network (for the simulations in section 5.3.2 and 5.3.3), it takes the form of 19 UDP flows of a constant bit-rate of 500 Mbps. Each flow has a randomly selected source and destination. These flows are injected into the network in a staggered fashion, starting every 200 ms, and each lasting until the end of the simulation. The simulations in section 5.3.2 and 5.3.3 differ in the domain from which the (source, destination) pairs are randomly chosen.

As a reference, *reliable BIER* (*i.e.,* using BIER for retransmissions, as per this chapter) is compared with multicast (as in *e.g.,* [128]) and unicast (as in *e.g.,* [129]) retransmissions of NACKed multicast data packets.

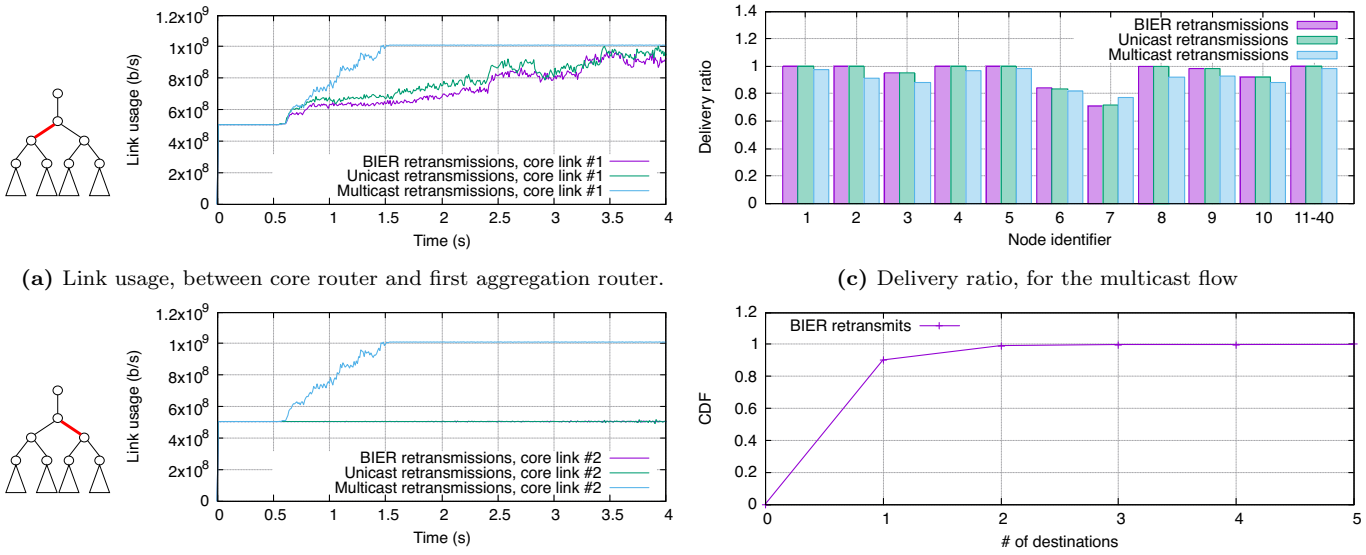
5.3.2 Uncorrelated, Localized Losses

For this set of simulations, UDP background flows are introduced with sources and destinations both within the leftmost rack (figure 5.3) as described in section 5.3.1. This will saturate some of the links between the ToR router and the individual machines, leading to packet losses in the "downwards" interfaces of the leftmost ToR router. BIER aggregation will thus only happen when, by chance, two or more clients detect a packet loss (and thus generate NACKs) at the same time.

Figure 5.4 depicts the results of a 4-second simulation run, specifically the usage of the two core links, the delivery ratio, and the distribution of the number of clients in BIER retransmissions. A first observation from figure 5.4b is that with multicast retransmissions, the rightmost aggregation link carries unnecessary traffic, unlike unicast and BIER retransmissions. It can also be observed that multicast retransmissions saturate the core links faster than the two other mechanisms: this is because excess retransmissions produce additional congestion, leading to additional losses of original transmissions, in turn leading to additional retransmissions.

Comparing with unicast retransmissions, the use of BIER retransmissions further minimizes the traffic footprint: even with uncorrelated losses, retransmissions are aggregated when several clients do not receive the same packet when using BIER. This is illustrated in figure 5.4d, which depicts the Cumulative Distribution Function (CDF) of the number of simultaneous clients to which a BIER retransmission is performed: $\approx 10\%$ of BIER retransmissions are destined for multiple (≥ 2) destinations, and thus benefit from aggregation. This allows a further reduction of link usage, as depicted in figure 5.4b.

A conclusion to draw from these simulations is that in case of localized losses, BIER and unicast retransmissions are preferable to multicast retransmissions – due to the latter incurring unnecessary traffic on links in paths unaffected by losses. Another conclusion is that when multiple



(a) Link usage, between core router and first aggregation router.

(c) Delivery ratio, for the multicast flow

(b) Link usage, between core router and second aggregation router.

(d) CDF of the number of clients in BIER retransmit bitstrings.

Figure 5.4 – Uncorrelated localized losses experiment. BIER retransmits vs unicast and multicast retransmits.

destinations do not receive a given multicast data packet, BIER retransmissions allow aggregation – an advantage over unicast retransmissions.

5.3.3 Correlated, Localized Losses

For this set of simulations, UDP background flows are introduced with sources and destinations within the **two** leftmost racks (figure 5.3) as described in section 5.3.1. This will, again, saturate some of the links – this time, in addition to between the individual machines and the ToR routers, also between the ToR routers and aggregation routers. A loss on one of these links will affect all the destinations within a rack.

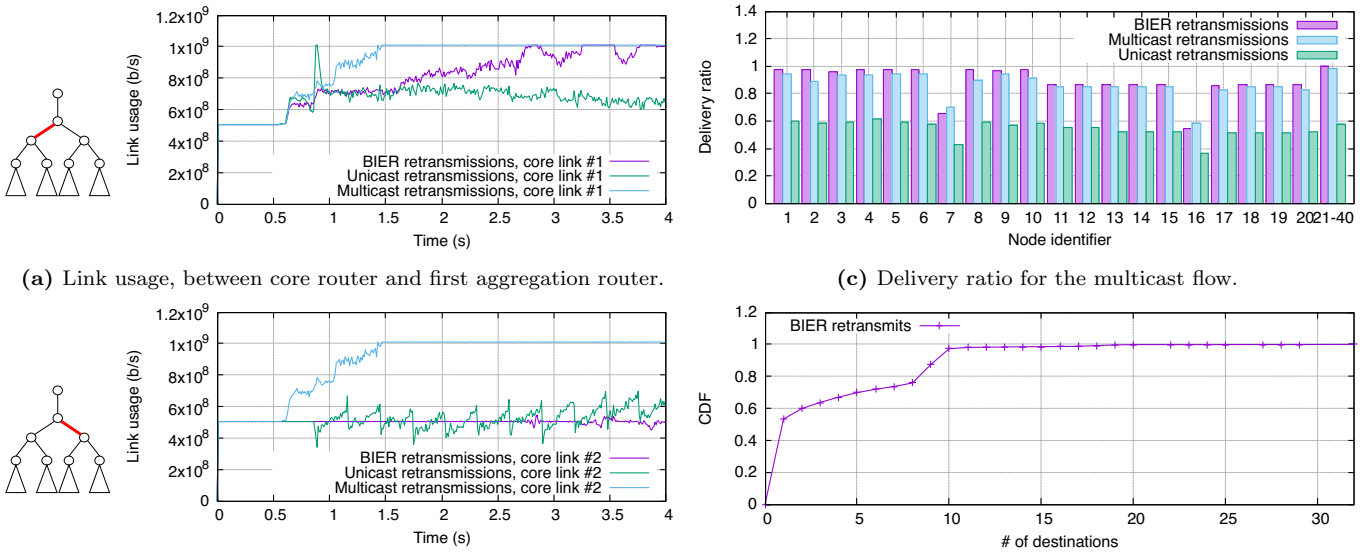
Figure 5.5 depicts the results of a 4-second simulation. For the same reasons as in section 5.3.2, multicast retransmissions cause unnecessary traffic on the right "half" of the data-center (see figure 5.3) – and cause earlier saturation on the left core link.

A single lost multicast data packet will, in this scenario, typically fail to be received by several destinations, therefore unicast retransmissions will generate a larger traffic footprint as compared to BIER retransmissions. With a link capacity of 1 Gbps and a multicast flow of 0.5 Gbps, the link between the source and the core router can sustain the unicast retransmission load only when each multicast data packet is, on average, retransmitted no more than once. Beyond that, the link becomes saturated with retransmissions, thus preventing "legitimate" original transmissions to succeed. This explains why unicast recovery incurs a lower delivery ratio, as depicted in figure 5.5c. This is also why the link usage on the first core link is lower for unicast retransmissions than for BIER retransmissions, as depicted in figure 5.5a.

The distribution of the number of simultaneous clients to which a multicast data packet is retransmitted is depicted in figure 5.5d: $\approx 46\%$ of BIER retransmissions are destined for multiple destinations, and thus benefit from aggregation.

5.3.4 Unlocalized, Bursty Losses

The simulations in sections 5.3.2 and 5.3.3 illustrate the benefits of BIER retransmissions when losses are spatially localized. Instead of creating background UDP flows, this section assumes uncontrolled, exogenous congestion in the data center – modelled by a Gilbert-Elliott loss model [148] on all links. This model is used to model bursty transmissions, and [149] shows that it accurately describes packet losses in the Internet. In sum, the Gilbert-Elliott loss model prescribes that a link can be in either a *good* or *bad* state. In *good* state, the link is ideal (no losses), whereas in *bad* state, the probability of a transmission to be successful (*i.e.,* to not be lost) is h. For each



(a) Link usage, between core router and first aggregation router.

(c) Delivery ratio for the multicast flow.

(b) Link usage, between core router and second aggregation router.

(d) CDF of the number of clients in BIER retransmit bitstrings.

Figure 5.5 – Correlated localized losses experiment. BIER retransmits vs unicast and multicast retransmits.

packet to be transmitted over a link, the state of the link may change: from *bad* to *good* with a probability of r, and from *good* to *bad* with a probability p.

For the purpose of the simulations in this section, the success probability in the *bad* state is set to $h = 0.5$, and the transition probability from *bad* to *good* to $r = 0.01$ (*i.e.,* the expected loss burst duration is 100 packets). The transition probability from *good* to *bad*, p, is set so that the average packet loss ratio is α, according to equation (4) in [149]: $p = \frac{r\alpha}{1-h-\alpha} = \frac{0.01\alpha}{0.5-\alpha}$. In order to quantify the sensitivity of the system to different congestion levels, 19 simulations are run, for $\alpha \in \{0.001, 0.002, \ldots, 0.01, 0.02, \ldots, 0.1\}$.

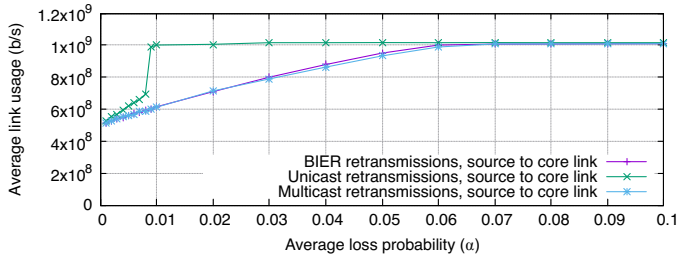
Figure 5.6 depicts the simulation results. A stability limit ($\alpha = 6\%$ for BIER and multicast, $\alpha = 0.9\%$ for unicast) can be observed in figure 5.6a. Above this limit, retransmissions compete with original transmissions over the link between the source and the core router, causing some of these original transmissions to be lost, thus requiring additional retransmissions. This then causes the delivery ratio to deteriorate, as depicted in figure 5.6c. Thus for this stability metric, *reliable BIER* shows superior results as compared to unicast retransmissions, while behaving similarly as compared to multicast retransmissions. Figure 5.6b shows the aggregate traffic (the sum of traffic induced by transmissions and retransmissions over all links) for each of the values of α. While unicast retransmissions (in the stability zone) behave slightly worse than BIER retransmissions, multicast retransmissions incur a substantial traffic footprint (of approximately 1.7× that of BIER retransmissions, for $\alpha = 6\%$).

A conclusion to draw from these simulations is that, when losses are unlocalized and bursty, BIER retransmissions are vastly preferable to multicast retransmissions, in terms of global traffic footprint and also vastly preferable to unicast retransmissions, in terms of avoiding saturation of individual links.

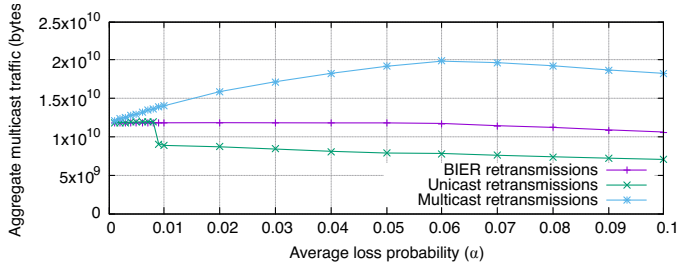
Finally, the CDF of the number of simultaneous clients to which a multicast data packet is retransmitted is depicted in figure 5.6d, which shows that when $\alpha \geq 6\%$, more than 50% of BIER retransmissions are destined for multiple destinations, and thus benefit from aggregation.

5.3.5 Influence of the Aggregation Timer

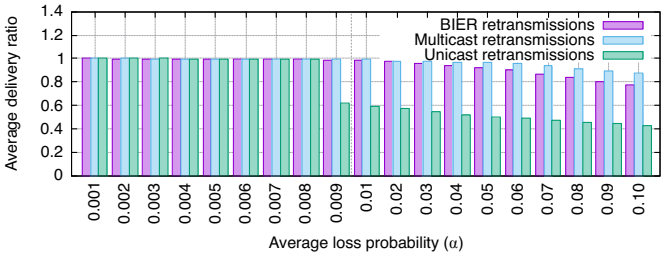
As described in section 5.2.3, increasing the aggregation timer Δt_{agg} (*i.e.,* the amount of time during which the source collects NACKs for a given packet before retransmitting) directly reduces the induced network traffic (since aggregating more NACKs means that the corresponding retransmission is sent to more clients), at the cost of packets being delivered later to the application. In order to quantify this phenomenon, an experiment using the scenario of section 5.3.4 is conducted. For two target loss probabilities $\alpha = 1\%$ and $\alpha = 0.8\%$ (slightly above and below the stability limit



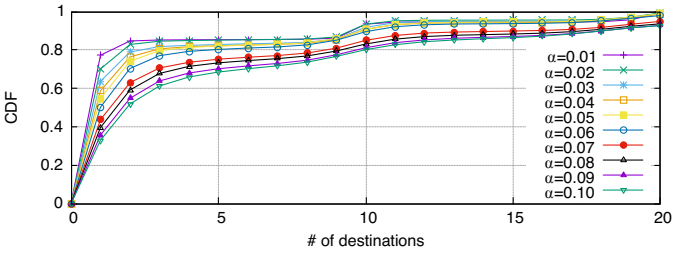
(a) Link usage, between the source and the core router (averaged over all the duration of a simulation).



(b) Aggregate multicast traffic (over all links and for all the duration of a simulation).



(c) Average delivery ratio for the multicast flow.



(d) CDF of the number of clients in BIER retransmit bitstrings.

Figure 5.6 – Unlocalized bursty losses experiments. 19 simulations with different loss probabilities.

observed in figure 5.6a, respectively), different values for the parameter Δt_{agg} are used, ranging from 25 μs to 10 ms.

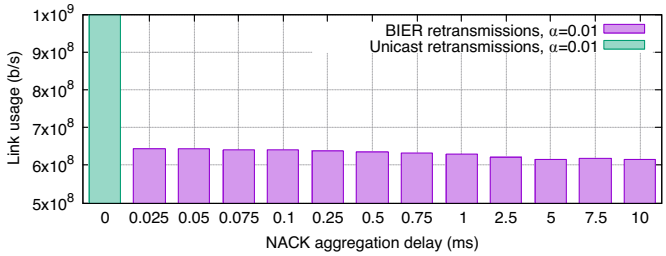
Figure 5.7a depicts the usage of the link between the source and the core router, averaged above the duration of the simulation, with $\alpha = 1\%$. It is interesting to observe that enabling NACK aggregation (*i.e.,* making $\Delta t_{agg} \neq 0$) causes a substantial improvement in performance. In particular, the use of any non-zero NACK aggregation delay leads to a significant benefit in terms of link usage: the smallest non-zero explored value ($\Delta t_{agg} = 25\ \mu s$) makes the link usage drop from 1 Gbps (with $\Delta t_{agg} = 0$) to 645 Mbps. Further increases of the aggregation delay lead to minor reductions of link usage (*e.g.,* from 645 Mbps for $\Delta t_{agg} = 25\ \mu s$, to 615 Mbps for $\Delta t_{agg} = 10\ ms$), at the cost of linearly increasing the delay of retransmitted packets by Δt_{agg}. Figure 5.7b depicts the results for $\alpha = 0.8\%$: a similar pattern can be observed, with a lower amount of traffic for unicast retransmissions – due to α being below the stability limit.

5.4 ISP Topology Simulations

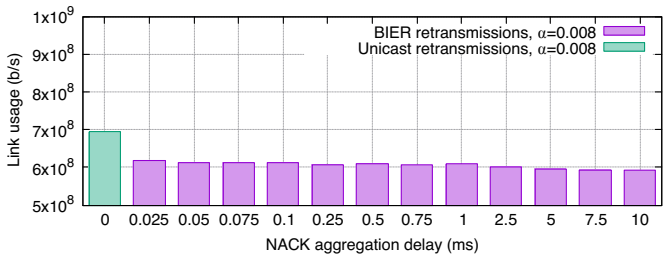
Sections 5.3.2, 5.3.3, and 5.3.4 illustrated the benefits of *reliable BIER* for reducing the traffic footprint in strict tree topologies such as those from data centers – begging the question of if these benefits are dependent on these topologies. In order to answer that question, this section presents simulations of a real topology, specifically that of BT Europe (Aug. 2010) from [150] (see figure 5.9). Note that while the topology used comes from a real deployment, this simulation does not claim to reproduce realistic Internet traffic: the goal is to explore the behavior of *reliable BIER*.

This topology consists of 24 routers, connected by 1 Gbps links. It is assumed that the unicast routing protocol has converged and each router has perfect shortest paths to all other routers. For the purpose of this simulation study, the source node is attached to router 17 (in London), and a destination is attached to each of the other 23 routers. The simulation parameters and multicast traffic flow parameters are as per section 5.3.1. Background traffic flows are also as per section 5.3.1, noting that router 17 is never chosen as source or destination for a background flow.

Figure 5.8a depicts the link usage between the source and router 17 – revealing that unicast retransmissions rapidly saturate the link, and that multicast retransmissions saturate the link faster than BIER retransmissions. Figure 5.8b depicts the link usage of the link between router 17 (to which the source is attached) and router 5 (one of its directly adjacent peer routers), noting that it is up to twice as high for multicast retransmissions as it is for both BIER retransmissions



(a) $\alpha = 1\%$



(b) $\alpha = 0.8\%$

Figure 5.7 – Influence of Δt_{agg}: link usage between the source and the core router.

and for unicast retransmissions. The link load for unicast retransmissions on this link is lower than for *reliable BIER*. The reason for this is that, as unicast retransmissions already saturated the link between the source and router 17 (figure 5.8a), fewer unicast retransmissions make it onto the link between router 17 and router 5 (figure 5.8b), causing an overall lower data delivery ratio when using unicast retransmissions. This is depicted in figure 5.8c, which also indicates a higher data delivery ratio for BIER retransmissions than for multicast retransmissions.

The distribution of the number of simultaneous clients to which a multicast data packet is retransmitted is depicted in figure 5.8d: $\approx 71\%$ of BIER retransmissions are destined for multiple destinations, and thus benefit from aggregation.

The conclusion to draw from these simulations is that the results obtained with a data-center topology (section 5.3) also can be valid for other topologies.

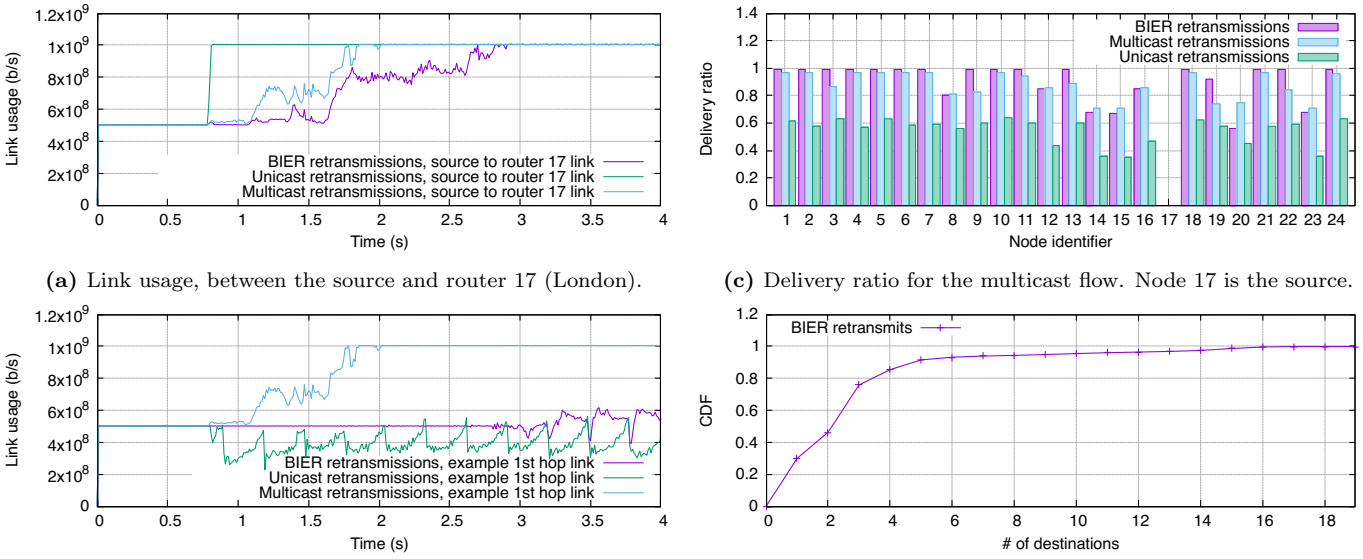
5.5 Reliable BIER Performance Analysis

The simulations in sections 5.3 and 5.4 illustrate the performance benefits of BIER-retransmissions for reliable multicast, both when faced with (i) rare, isolated losses, and with (ii) correlated, frequent losses in traffic-intensive environments. Specifically, *reliable BIER* was observed to result in a substantially lower traffic footprint in the simulated scenarios, with equivalent or better multicast data packet delivery ratios than when using multicast and unicast retransmissions.

This section aims at generalizing these observations by way of formulating an analytical model of arbitrary tree topologies, and of using this model to analytically quantify the number of successful and failed transmissions[4] necessary for a reliable multicast operation to succeed (*i.e.,* for all destinations to have received a copy of a multicast data packet). Section 5.5.2 derives an exact expression of this as $M^{\mathrm{B}}_{[i]}$, for *reliable BIER* – and for comparison, $M^{\mathrm{m}}_{[i]}$ and $M^{\mathrm{u}}_{[i]}$ for when using multicast and unicast retransmissions, respectively.

These exact expressions, however, become mathematically intractable for large trees, thus section 5.5.3 develops a first-order approximation of the average traffic footprints of reliable multicast using BIER, multicast, and unicast retransmissions, respectively.

[4] Colloquially speaking, to *count the blue arrows* in figure 5.1.



(a) Link usage, between the source and router 17 (London).

(c) Delivery ratio for the multicast flow. Node 17 is the source.

(b) Link usage, between router 17 (London) and router 5.

(d) CDF of the number of clients in BIER retransmit bitstrings.

Figure 5.8 – ISP topology experiment. BIER retransmits vs unicast and multicast retransmits.

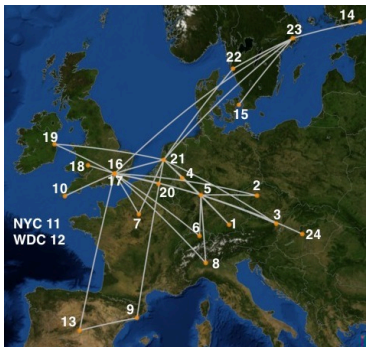


Figure 5.9 – Network topology (picture from [150]).

5.5.1 Model, Assumptions and Definitions

Network links have an associated packet loss probability[5] $\alpha \in [0, 1]$. As illustrated in section 5.1, multicast transmissions and retransmissions (regardless of if BIER, multicast, or unicast retransmissions) span a tree, rooted in the source. For describing these trees, the following notation is introduced: routers and destinations are indiscriminately termed *node*, and each node is uniquely labeled with the path from the root of the tree to it[6]; $[[i], j]$ denotes the j-th child of node $[i]$, and the term *"the subtree $[i]$"* refers to the subtree which is rooted in node $[i]$. Finally, the set of children of $[i]$ is denoted $c([i])$: $c([i]) = \{[[i], 1], [[i], 2], \ldots\}$

This analysis assumes retransmissions by the source until all destinations have received a copy of the multicast data packet, and that the source collects all generated NACKs before retransmitting a packet (*i.e.*, $\ell = \infty$, no NACKs are lost, $\Delta t_{agg} \geq RTT_{max} - RTT_{min}$, $\Delta t_{retry} \geq \Delta t_{agg} + RTT_{max}$). It quantifies, under these assumptions, (1) the number of retransmissions of a multicast data packet that are made by the source, and (2) the total number of transmissions in the network, until all clients have received (at least) one copy of the multicast data packet.

[5] For lossless links, operating below capacity and with finite buffers, packet losses are due to buffer overflow – thus while a link may be lossless, an interface may still experience packet losses.

[6] The root is labelled $[1]$; the first child of the root is labeled $[[1], 1]$, its second child $[[1], 2]$; the first child of $[[1], 1]$ is $[[[1], 1], 1]$; etc.

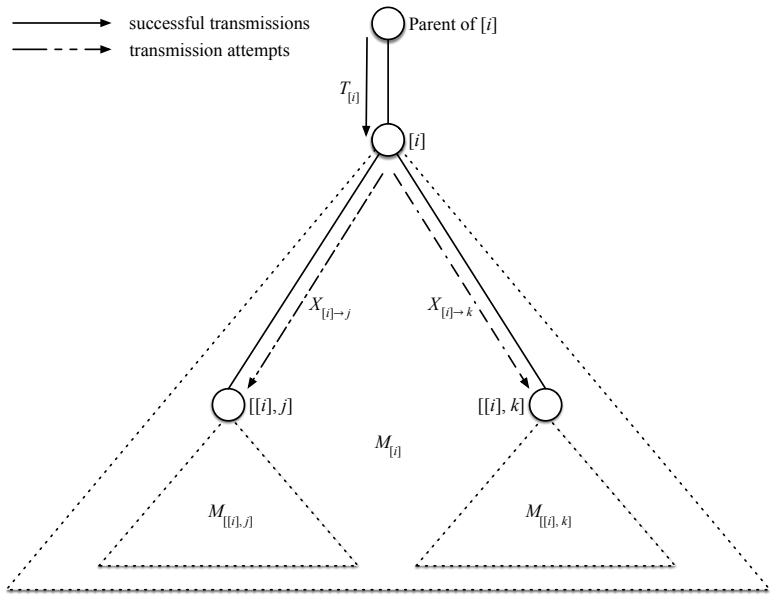


Figure 5.10 – Notation

Definitions:

Given a node $[i]$ and its child $[[i], j]$, and with reference to figure 5.10:

- $T_{[[i],j]}$: is the number of *attempts* from node $[[i], j]$, *i.e.,* number of times that $[[i], j]$ must transmit copies of a multicast data packet to its children so as to ensure that all destinations in its subtree receive the multicast data packet. If $[[i], j]$ is a leaf, then by convention $T_{[[i],j]} = 1$.
- $X_{[i]\to j}$: is the number of *transmissions* made by $[i]$ over the link $([i], [[i], j])$, needed to ensure that node $[[i], j]$ receives the $T_{[[i],j]}$ copies of the multicast data packet.
- $M^{\star}_{[i]}$: is the number of *packets* transmitted inside a subtree $[i]$ to ensure that all destinations receive a copy, where $\star$ indicates the considered variant (B for BIER, m for multicast, u for unicast). If $[i]$ is a leaf, then by convention $M^{\star}_{[i]} = 0$.

The number of attempts by $[i]$ is the worst of the number of transmissions on all links $([i], [[i], j])$:

$$T_{[i]} = \max_{[[i],j]\in c([i])} X_{[i]\to j} \tag{5.1}$$

5.5.2 Computation of $T_{[i]}$, $X_{[i]\to j}$, and $M^{\star}_{[i]}$

Each node $[[i], j]$ needs to receive $T_{[[i],j]}$ copies of the multicast data packet from its parent, $[i]$. For each of these, the number of transmissions over the link $([i], [[i], j])$ until the copy is successfully received at $[[i], j]$ is geometrically distributed with success probability $(1 - \alpha)$, which leads to the following proposition:

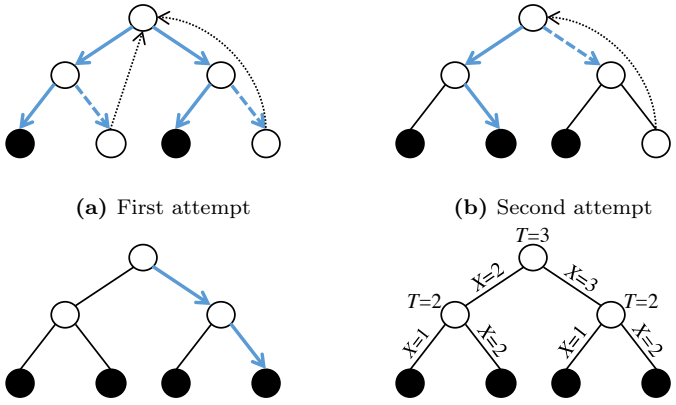
Proposition 5.1. *The total number of transmissions over the link* $([i], [[i], j])$ *follows a negative binomial distribution with (random) parameter* $T_{[[i],j]}$*. For* $x \geq k \geq 1$*:*

$$\mathbf{P}[X_{[i]\to j} = x | T_{[[i],j]} = k] = \binom{x-1}{k-1} \alpha^{x-k} (1-\alpha)^k \tag{5.2}$$

and the number of attempts from node $[i]$*,* $T_{[i]}$ *is:*

$$\mathbf{P}[T_{[i]} = k] = \prod_{[[i],j]\in c([i])} \mathbf{P}[X_{[i]\to j} \leq k] - \prod_{[[i],j]\in c([i])} \mathbf{P}[X_{[i]\to j} \leq k-1] \tag{5.3}$$

Equations (5.2) and (5.3) allow computing the probability density function (PDF) for $X_{[i]\to j}$ and $T_{[i]}$ recursively from the leaves towards the root, using the convention that $T_{[i]} = 1$ for a leaf.



(a) First attempt **(b)** Second attempt

(c) Third attempt **(d)** Total number of attempts and transmissions

Figure 5.11 – *Reliable BIER* examplified – solid blue arrows represent successful transmissions, dashed blue arrows represent unsuccessful transmissions, and dashed black arrows represent NACKs. A NACK is sent by two nodes (a) upon receipt of a subsequent packet in the stream. After retransmission (b), the last node still has not received a copy of the packet, and sends a second NACK upon timeout of Δt_{retry} (Algorithm 6). The third attempt (c) ensures that delivery is successful. Figure (d) shows the total number of *attempts* $T_{[i]}$ at each node $[i]$, and total *transmissions* $X_{[i]\to j}$ at each link $[i] \to [[i], j]$. The total number of packets sent in the tree (the sum of all $X_{[i]\to j}$) is $M^{\mathrm{B}}_{[i]} = 11$.

BIER retransmissions: $M^{B}_{[i]}$

When using BIER retransmissions, $M^{\mathrm{B}}_{[i]}$ is the sum of the transmissions on each link $([i], [[i], j])$ and the packets transmitted in each subtree $[[i], j]$, *i.e.,* :

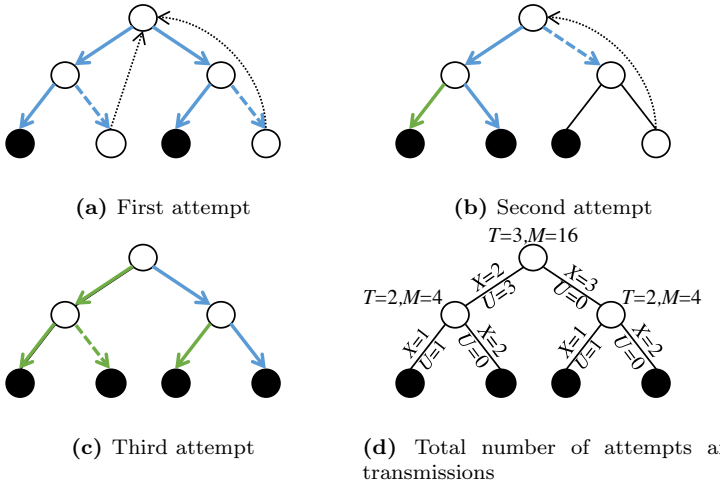
$$M^{\mathrm{B}}_{[i]} = \sum_{[[i],j]\in c([i])} (X_{[i]\to j} + M^{\mathrm{B}}_{[[i],j]}) \tag{5.4}$$

Figure 5.11 provides a detailed example of a BIER reliable transmission, with the corresponding values for $T_{[i]}$, $X_{[i]\to j}$ and $M^{\mathrm{B}}_{[i]}$ displayed in Fig. 5.11d.

Multicast retransmissions: $M^{m}_{[i]}$

The number of multicast data packets sent over a network with multicast retransmissions can be obtained by adapting the previously presented model (section 5.5.2). Consider the transmission from a node $[i]$ to a node $[[i], j]$: transmitted packets can be classified into two categories: (i) packets sent until the subtree $[[i], j]$ is covered, and (ii) packets flooded by $[i]$ inside the subtree $[[i], j]$ after it has been covered. The latter packets come from retransmissions from the source $[i]$ that are due to other subtrees $[[i], k]$ having not yet been covered. Let $U_{[i]\to[[i],j]}$ be the number of packets that fall into the second category. The number of floods is $T_{[i]} - X_{[i]\to j}$ (*i.e.,* the number of times $[i]$ transmits after $[[i], j]$ has already received enough packets): index these floods with $f \in [1, T_{[i]} - X_{[i]\to j}]$. For each of these floods, let $Y^{f}_{[i]\to j}$ be a Bernoulli variable of parameter $(1 - \alpha)$ representing the success of transmission on the link $[i] \to [[i], j]$, and $F^{f}_{[[i],j]}$ be a variable representing the number of packets flooded in the subtree $[[i], j]$. The number of unnecessary packets is then, for each of these floods, one packet (from $[i]$ to $[[i], j]$) plus, if the transmission succeeded (*i.e.,* if $Y^{f}_{[i]\to j} = 1$), the number of packets $F^{f}_{[[i],j]}$ resulting from a multicast flood sourced at $[[i], j]$:

$$U_{[i]\to[[i],j]} = \sum_{f=1}^{T_{[i]}-X_{[i]\to j}} (1 + Y^{f}_{[i]\to j} F^{f}_{[[i],j]}) \tag{5.5}$$



(a) First attempt **(b)** Second attempt

(c) Third attempt **(d)** Total number of attempts and transmissions

Figure 5.12 – Example of multicast reliable transmission. In addition to the conventions of figure 5.11, green arrows represent unnecessary retransmissions. In the second attempt (b), node $[[1],1]$ floods one unnecessary packet. In the third attempt (c), the root floods three unnecessary packets, and node $[[1],2]$ floods one unnecessary packet. Figure (d) shows the total number of *attempts* $T_{[i]}$ at each node $[i]$, total *transmissions* $X_{[i]\to j}$ at each link $[i] \to [[i],j]$, and the total number of *unnecessary packets* $U_{[i]\to[[i],j]}$ flooded by each node $[i]$ in the subtree $[[i],j]$. The total number of packets sent in the tree (the sum of all $X_{[i]\to j}$ and $U_{[i]\to[[i],j]}$) is $M^{\mathrm{m}}_{[1]} = 16$.

where the mean number of packets sent in a multicast flood from $[i]$, $\mathbf{E}[F_{[j]}]$, can be recursively computed as follows (with $\mathbf{E}[F_{[j]}] = 0$ for every leaf $[j]$):

$$\mathbf{E}[F_{[i]}] = \sum_{[[i],j]\in c([i])} (1 + (1-\alpha)\mathbf{E}[F_{[[i],j]}]) \tag{5.6}$$

From this, the total number of packets sent in a subtree $[i]$ until all of its destinations receive a copy, $M^{\mathrm{m}}_{[i]}$, can be computed recursively. It corresponds, for each child $[[i],j]$, to the number of transmissions over the link $[i] \to [[i],j]$ required by $[[i],j]$, plus the number of packets sent inside $[[i],j]$ so as to cover all destinations, plus the unnecessary multicast packets originating from $[i]$:

$$M^{\mathrm{m}}_{[i]} = \sum_{[[i],j]\in c([i])} \left[X_{[i]\to j} + M^{\mathrm{m}}_{[[i],j]} + U_{[i]\to[[i],j]}\right] \tag{5.7}$$

Proposition 5.2 indicates a simple way to compute the average traffic footprint for multicast retransmissions, using $T_{[i]}$ and $F_{[i]}$.

Proposition 5.2. *Let $[i]$ be a node in the tree. With multicast retransmissions, the mean number of packets sent until all destinations in $[i]$ obtain a copy can be computed as such:*

$$\mathbf{E}[M^{m}_{[i]}] = \mathbf{E}[T_{[i]}]\mathbf{E}[F_{[i]}] \tag{5.8}$$

Proof. by induction. If $[i]$ is a leaf, $M^{\mathrm{m}}_{[i]} = 0$ and $F_{[i]} = 0$, hence the result holds. Otherwise, let $[i]$ be a node that is not a leaf, and assume that the result holds for all children of $[i]$. Then, using equation (5.7), and Wald's equation to expand $\mathbf{E}[U_{[i]\to[[i],j]}]$ from equation (5.5), it follows that:

$$
\begin{aligned}
\mathbf{E}[M^{\mathrm{m}}_{[i]}] &= \sum_{[[i],j]\in c([i])} \Big[\mathbf{E}[X_{[i]\to j}] + \mathbf{E}[M^{\mathrm{m}}_{[[i],j]}] + \mathbf{E}[T_{[i]} - X_{[i]\to j}](1 + (1-\alpha)\mathbf{E}[F_{[[i],j]}])\Big] \\
&= \sum_{[[i],j]\in c([i])} \Big[\mathbf{E}[X_{[i]\to j}] + \mathbf{E}[T_{[[i],j]}]\mathbf{E}[F_{[[i],j]}] \\
&\qquad\qquad + \mathbf{E}[T_{[i]} - X_{[i]\to j}](1 + (1-\alpha)\mathbf{E}[F_{[[i],j]}])\Big] \\
&= \sum_{[[i],j]\in c([i])} \Big[\mathbf{E}[X_{[i]\to j}] + (1-\alpha)\mathbf{E}[X_{[i]\to j}]\mathbf{E}[F_{[[i],j]}] \\
&\qquad\qquad + \mathbf{E}[T_{[i]} - X_{[i]\to j}](1 + (1-\alpha)\mathbf{E}[F_{[[i],j]}])\Big] \\
&= \sum_{[[i],j]\in c([i])} \mathbf{E}[T_{[i]}](1 + (1-\alpha)\mathbf{E}[F_{[[i],j]}]) \\
&= \mathbf{E}[T_{[i]}]\mathbf{E}[F_{[i]}]
\end{aligned}
$$

where the equality $\mathbf{E}[X_{[i]\to j}] = \frac{\mathbf{E}[T_{[[i],j]}]}{1-\alpha}$ and equation (5.6) were used. □

Figure 5.12 provides a detailed example of multicast reliable transmission, with corresponding values for the variables $T_{[i]}$, $X_{[i]\to j}$, $U_{[i]\to[[i],j]}$, and $M^{\mathrm{m}}_{[i]}$.

Unicast retransmissions: $M^{u}_{[i]}$

With unicast retransmissions, losses experienced by each destination are treated individually by the source. Given the loss of a multicast data packet (sent by the source [1]) at a destination $[c]$, connected to the source in $d([c])$ hops, the number of retransmissions from the source before successful delivery of the packet to $[c]$ is a random variable, $R_{[1]\to[c]}$, whose mean is described in proposition 5.3.

Proposition 5.3. *The mean value of* $R_{[1]\to[c]}$ *is:*

$$\mathbf{E}[R_{[1]\to[c]}] = \frac{1-(1-\alpha)^{d([c])}}{\alpha(1-\alpha)^{d([c])}} \tag{5.9}$$

Proof. Let $[c]$ be a destination at depth $d([c])$; for simplicity, write $d = d([c])$. The unicast retransmission will succeed if the packet successfully traverses d successive links: the probability of a unicast success from the source to $[c]$ is therefore $(1-\alpha)^d$. Let $Z_{[1]\to[c]}$ represent the number of trials before (and not counting) the unicast success. $Z_{[1]\to[c]}$ is geometrically distributed with parameter $(1-\alpha)^d$:

$$
\begin{aligned}
\mathbf{P}[Z_{[1]\to[c]} = k] &= (1-\alpha)^d[1-(1-\alpha)^d]^k, \forall k \geq 0 \\
\mathbf{E}[Z_{[1]\to[c]}] &= \frac{1-(1-\alpha)^d}{(1-\alpha)^d}
\end{aligned}
$$

For each of these first $Z_{[1]\to[c]}$ (unsuccessful) attempts, $N_{[1]\to[c]}$ unicast packets will be sent through the chain of links from [1] to $[c]$, where $N_{[1]\to[c]}$ is distributed as:

$$
\begin{aligned}
\mathbf{P}[N_{[1]\to[c]} = k] &= \frac{(1-\alpha)^{k-1}\alpha}{1-(1-\alpha)^d}, \forall 1 \leq k \leq d \\
\mathbf{E}[N_{[1]\to[c]}] &= \frac{1-\alpha d(1-\alpha)^d - (1-\alpha)^d}{\alpha(1-(1-\alpha)^d)}
\end{aligned}
$$

The last (successful) unicast attempt will generate d packets (one per link). Hence, the total number of unicast packets sent until the destination $[c]$ receives a copy, $R_{[1]\to[c]}$, is:

$$\mathbf{E}[R_{[1]\to[c]}] = \mathbf{E}[Z_{[1]\to[c]}]\mathbf{E}[N_{[1]\to[c]}] + d = \frac{1-(1-\alpha)^d}{\alpha(1-\alpha)^d}$$

as desired. □

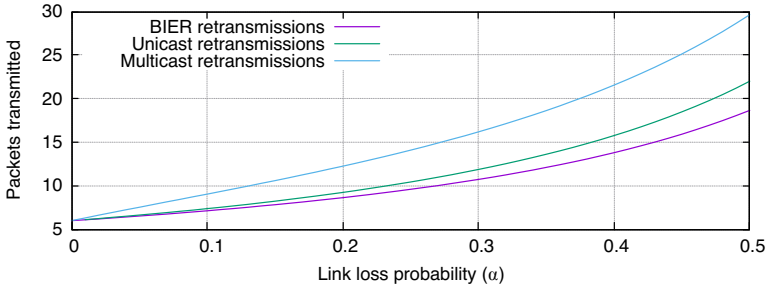


Figure 5.13 – Number of packets $M^{\mathrm{B}}_{[1]}$ (BIER retransmissions), $M^{\mathrm{m}}_{[1]}$ (multicast retransmissions) and $M^{\mathrm{u}}_{[1]}$ (unicast retransmissions) transmitted in the binary tree of figure 5.1 until all destinations receive a copy.

The previous proposition allows to compute the total number of multicast data packets sent from the source with the unicast reliability mechanism, $M^{\mathrm{u}}_{[1]}$. This variable corresponds to the multicast data packets sent in the first multicast flood, plus, for each destination that did not receive a copy of the multicast data packet, the number of unicast retransmissions needed until the copy is successfully received. Proposition 5.4 expresses $M^{\mathrm{u}}_{[1]}$ and provides a closed expression for its mean, $\mathbf{E}[M^{\mathrm{u}}_{[1]}]$.

Proposition 5.4. *Let $\mathcal{C}$ be the set of destinations, and let $\mathcal{F}_{[1]}$ be the (random) set of destinations that have successfully received a copy after the first multicast flood by* [1]*. Then, the number of multicast data packets sent from the source, under unicast retransmissions, until each destination has received a copy is:*

$$M^{u}_{[1]} = F_{[1]} + \sum_{[c]\in\mathcal{C}} \mathbf{1}_{\{[c]\notin\mathcal{F}_{[1]}\}} R_{[1]\to[c]} \tag{5.10}$$

and its mean is:

$$\mathbf{E}[M^{u}_{[1]}] = \mathbf{E}[F_{[1]}] + \sum_{[c]\in\mathcal{C}} \frac{(1-(1-\alpha)^{d([c])})^2}{\alpha(1-\alpha)^{d([c])}} \tag{5.11}$$

Proof. From the definition of $\mathcal{F}_{[1]}$, equation (5.10) holds. The mean number of multicast data packets sent in the network is therefore:

$$\mathbf{E}[M^{\mathrm{u}}_{[1]}] = \mathbf{E}[F_{[1]}] + \sum_{[c]\in\mathcal{C}} \mathbf{P}[[c]\notin\mathcal{F}_{[1]}]\mathbf{E}[R_{[1]\to[c]}]$$

And since $\mathbf{P}[[c]\notin\mathcal{F}_{[1]}] = 1-(1-\alpha)^{d([c])}$, the result in (5.11) is obtained by using equation (5.9). □

Computational Example

For trees of small height, it is possible to recursively derive the average number of transmissions needed in the network in order to deliver a multicast data packet to all destinations, in an exact manner, for BIER reliability (proposition 5.1 and equation (5.4)), multicast reliability (proposition 5.2) or unicast reliability (proposition 5.4). As an example, figure 5.13 reports the results of this computation, for a binary tree of height 2 (as in figure 5.1): as expected, reliable BIER incurs the lowest overhead.

5.5.3 Total Traffic Approximation

Directly computing the traffic footprint is intractable when the depth of the tree is important. Therefore, this section provides a first-order approximation of the expected number of packets $\mathbf{E}[M^{\star}_{[i]}]$ transmitted in the network before all destinations receive a copy, under the assumption that losses are rare (*i.e.,* $\alpha \to 0$, as in [143]), for arbitrary trees and for each retransmission

mechanism (reliable BIER, unicast retransmissions, multicast retransmissions). This generalizes the model provided in [143], which provides a first-order approximation of the expected number of source-transmissions $\mathbf{E}[T_{[i]}]$, for multicast retransmissions.

For a link l of the tree, $d(l)$ is the depth of l ($d(l) = 1$ for a link rooted at the source); L is the total number of links in the tree, and D is the average depth of a links in the tree: $D = \frac{1}{L}\sum_l d(l)$. C is the number of destinations, and Δ is the average squared depth of a destination: $\Delta = \frac{1}{C}\sum_c d(c)^2$. Given these parameters, Theorem 5.1 describes first-order (for α) approximations of the number of transmissions in the network:

Theorem 5.1. *The average number of multicast data packets that need to be sent in the tree until all destinations have received a copy are, for reliable BIER ($M^B_{[1]}$), for multicast retransmissions ($M^m_{[1]}$) and for unicast retransmissions ($M^u_{[1]}$), given by the following approximations when $\alpha \to 0$:*

$$\begin{cases} \mathbf{E}[M^B_{[1]}] = L + LD\alpha + \mathcal{O}(\alpha^2) \\ \mathbf{E}[M^m_{[1]}] = L + [L^2 - L(D-1)]\alpha + \mathcal{O}(\alpha^2) \\ \mathbf{E}[M^u_{[1]}] = L + [C\Delta - L(D-1)]\alpha + \mathcal{O}(\alpha^2) \end{cases} \tag{5.12}$$

Proof. The proof is deferred to section 5.5.5. □

When there are no losses ($\alpha = 0$), the transmission of one multicast data packet yields L packets in the tree (one per link), whichever retransmission mechanism (BIER, multicast or unicast) is used. In addition to these L packets, with BIER retransmissions, the traffic is approximately $LD\alpha$ packets, as compared to approximately $L^2\alpha$ packets for multicast retransmissions and approximately $C\Delta\alpha$ for unicast retransmissions (Theorem 5.1). The traffic due to unicast or multicast retransmissions can thus be orders of magnitudes bigger than the corresponding BIER traffic, if the number of links and/or the depth of destinations is important.

When the tree is *regular* (*i.e.,* each node has the same number of children), it is possible to further simplify these expressions. Let c be the "arity" of the tree, defined as the number of children per node (for instance, a binary tree has $c = 2$), and h be the height of the tree (where a tree of height 0 is defined to have a single node). With these notations, it is possible to derive the following approximations (when the arity c is fixed) for $L \to \infty$:

$$\begin{cases} \lim_{\alpha\to 0} \frac{1}{\alpha}\mathbf{E}[M^{\mathrm{B}}_{[1]} - L] = \Theta(L \log L) \\ \lim_{\alpha\to 0} \frac{1}{\alpha}\mathbf{E}[M^{\mathrm{u}}_{[1]} - L] = \Theta(L \log^2 L) \\ \lim_{\alpha\to 0} \frac{1}{\alpha}\mathbf{E}[M^{\mathrm{m}}_{[1]} - L] = \Theta(L^2) \end{cases} \tag{5.13}$$

This provides a hierarchy between the three studied mechanisms, and shows that BIER retransmissions reduce the traffic footprint from a quadratic behavior in the number of links (with multicast retransmissions) to a log-linear behavior.

Proof. The number of links in the tree can be expressed as $L = \sum_{k=1}^{h} c^k = \frac{c^{h+1}-c}{c-1}$. This allows expressing h as a function of L:

$$h = \frac{1}{\log c} \log\left(1 + L\frac{c-1}{c}\right) = \frac{\log L}{\log c} + \mathcal{O}(1)$$

Then, the average depths of links in the tree is:

$$\begin{aligned} D &= \frac{1}{L}\sum_{k=1}^{h} kc^k = \frac{c-1}{c^{h+1}-c} \times \frac{hc^{h+2} - (h+1)c^{h+1} + c}{(c-1)^2} \\ &= h - \frac{1}{c-1} + \frac{h}{c^h - 1} = h - \underbrace{\frac{1}{c-1} + \frac{c}{c-1}\frac{h}{L}}_{\mathcal{O}(1)} \\ &= \frac{\log L}{\log c} + \mathcal{O}(1) \end{aligned}$$

which yields $LD = \Theta(L \log L)$, as desired to obtain the result for BIER retransmissions.

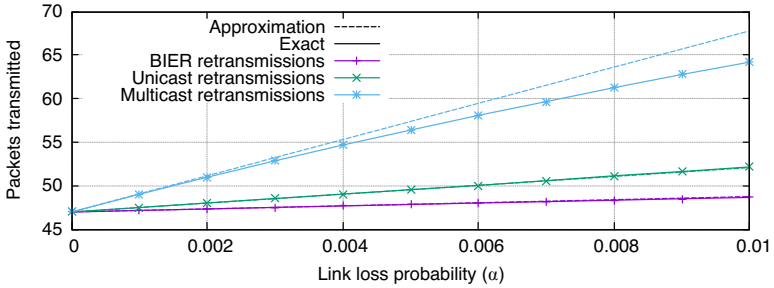


Figure 5.14 – Number of packets $M^{\mathrm{B}}_{[1]}$ (BIER retransmissions), $M^{\mathrm{m}}_{[1]}$ (multicast retransmissions) and $M^{\mathrm{u}}_{[1]}$ (unicast retransmissions) transmitted in the tree of figure 5.3 until all destinations receive a copy. Solid lines represent exact values (obtained by simulation), dotted lines represent the low-loss approximation given in theorem 5.1.

k	L	$L \times D$	C	BIER	unicast	multicast
4	28	68	16	68α	104α	744α
6	78	204	54	204α	360α	5958α
8	168	456	128	456α	864α	27936α
10	310	860	250	860α	1700α	95550α
12	516	1452	432	1452α	2952α	265320α
14	798	2268	686	2268α	4704α	635334α
16	1168	3344	1024	3344α	7040α	1362048α

Table 5.1 – Average number of retransmissions per multicast data packet for $(k, k/2, k/2)$ tree topologies (approximation as per theorem 5.1)

For multicast retransmissions, since $LD = \Theta(L \log L)$, this yields:

$$L^2 - L(D-1) = \Theta(L^2)$$

as desired.

Finally, for unicast retransmissions, the sum of the squared depths of clients is:

$$\begin{aligned} C\Delta &= h^2 c^h = \left(\frac{\log L}{\log c} + \mathcal{O}(1)\right)^2 \left(1 + L\frac{c-1}{c}\right) \\ &= \frac{c-1}{c \log^2 c} L \log^2 L + \mathcal{O}(L \log L) = \Theta(L \log^2 L) \end{aligned}$$

and since $LD = \Theta(L \log L)$, this finally yields:

$$C\Delta - L(D-1) = \Theta(L \log^2 L)$$

as desired, thus concluding the proof. □

5.5.4 Discussion

The accuracy and relevance of approximations from theorem 5.1 can be assessed against simulations in realistic tree topologies. Two examples are examined in this section: (1) reliable multicast over a datacenter-like topology as depicted in figure 5.3, and (2) multicast flows over fat-tree-like topologies (introduced in figure 1.3).

The datacenter-like topology of figure 5.3 yields the parameters $L = 47$, $C = 40$, $D = \frac{177}{47}$, and $\Delta = 16$. When $\alpha \to 0$, retransmissions will incur a footprint of 177α packets for BIER, 510α packets for unicast, and 2079α packets for multicast. In order to quantify the quality of the approximation for this example, the means for $M^{\mathrm{B}}_{[1]}$, $M^{\mathrm{m}}_{[1]}$ and $M^{\mathrm{u}}_{[1]}$ have been computed over 10^6 random samples, for different values of α with $0 \leq \alpha \leq 1\%$. Figure 5.14 depicts the results of these simulations, as well as the linear approximation from theorem 5.1. For BIER and unicast retransmissions, the approximation accurately fits the computed mean. For multicast retransmissions,

the approximation is within a 6% error margin of the computed value.

For reliable multicast flows over fat-tree-like topologies, the root has k children, each having $k/2$ children, each also having $k/2$ children. For these trees, the parameters become: $L = k + \frac{k^2}{2} + \frac{k^3}{4}$, $L \times D = k + k^2 + \frac{3k^3}{4}$, $C = \frac{k^3}{4}$, $\Delta = 9$, allowing calculating the approximation of theorem 5.1. Table 5.1 depicts the approximate retransmission footprint for BIER, multicast and unicast retransmissions. It can be observed that unicast retransmissions exhibit a footprint approximately twice as high as BIER retransmissions. The footprint for multicast retransmissions is at least one order of magnitude higher, and clearly does not scale with the number of clients.

5.5.5 Proof of Theorem 5.1

This section provides a proof of theorem 5.1 and can be skipped at first reading. For this, three lemmas will be needed. Lemma 5.1 first gives an approximation at order 1 in α of the probabilities of having one or two transmissions. Then, lemma 5.2 gives a bound on the corresponding probability distributions, which will be used in lemma 5.3 to show that terms corresponding to three or more transmissions do not contribute to the terms of order 1 in α.

For an arbitrary node $[i]$ in the tree, let $l([i])$ be the number of links in the subtree rooted at $[i]$, with $l([i]) = 0$ if $[i]$ is a leaf.

Lemma 5.1. *Let $[i]$ be a node in the tree, and (if $[i]$ is not a leaf) $[[i], j]$ be an arbitrary child of $[i]$. The following approximations hold when $\alpha \to 0$:*

$$\begin{aligned}
\mathbf{P}[X_{[i]\to j} = 1] &= 1 - (1 + l([[i], j]))\alpha + \mathcal{O}(\alpha^2) \\
\mathbf{P}[X_{[i]\to j} = 2] &= (1 + l([[i], j]))\alpha + \mathcal{O}(\alpha^2) \\
\mathbf{P}[T_{[i]} = 1] &= 1 - l([i])\alpha + \mathcal{O}(\alpha^2) \\
\mathbf{P}[T_{[i]} = 2] &= l([i])\alpha + \mathcal{O}(\alpha^2)
\end{aligned}$$

Proof. by induction over the structure of the tree. If $[i]$ is a leaf, then $T_{[i]} = 1$ by definition (a client needs one copy of the packet). Otherwise, let $[i]$ be a node that is not a leaf, and assume that the result holds for all children of $[i]$. Let $[[i], j]$ be an arbitrary child of $[i]$. Equation (5.2) yields:

$$\begin{aligned}
\mathbf{P}[X_{[i]\to j} = 1] &= \mathbf{P}[T_{[[i],j]} = 1](1 - \alpha) \\
&= (1 - l([[i], j])\alpha + \mathcal{O}(\alpha^2))(1 - \alpha) \\
&= 1 - (1 + l([[i], j]))\alpha + \mathcal{O}(\alpha^2) \\
\mathbf{P}[X_{[i]\to j} = 2] &= \mathbf{P}[T_{[[i],j]} = 1]\alpha(1 - \alpha) + \mathbf{P}[T_{[[i],j]} = 2](1 - \alpha)^2 \\
&= (1 - l([[i], j])\alpha + \mathcal{O}(\alpha^2))\alpha(1 - \alpha) + (l([[i], j])\alpha + \mathcal{O}(\alpha^2))(1 - \alpha)^2 \\
&= (1 + l([[i], j]))\alpha + \mathcal{O}(\alpha^2)
\end{aligned}$$

Then, the definition of $T_{[i]}$ gives:

$$\begin{aligned}
\mathbf{P}[T_{[i]} = 1] &= \prod_{[[i],j]\in c([i])} \mathbf{P}[X_{[i]\to j} = 1] \\
&= \prod_{[[i],j]\in c([i])} \left[1 - (1 + l([[i], j]))\alpha + \mathcal{O}(\alpha^2)\right] \\
&= 1 - \sum_{[[i],j]\in c([i])} \left[1 + l([[i], j])\right] \alpha + \mathcal{O}(\alpha^2) \\
&= 1 - l([i])\alpha + \mathcal{O}(\alpha^2) \\
\mathbf{P}[T_{[i]} = 2] &= \prod_{[[i],j]\in c([i])} \mathbf{P}[X_{[i]\to j} \leq 2] - \prod_{[[i],j]\in c([i])} \mathbf{P}[X_{[i]\to j} \leq 1] \\
&= \prod_{[[i],j]\in c([i])} (1 + \mathcal{O}(\alpha^2)) - \prod_{[[i],j]\in c([i])} \left[1 - (1 + l([[i], j]))\alpha + \mathcal{O}(\alpha^2)\right] \\
&= \sum_{[[i],j]\in c([i])} \left[1 + l([[i], j])\right] \alpha + \mathcal{O}(\alpha^2)
\end{aligned}$$

$$= l([i])\alpha + \mathcal{O}(\alpha^2)$$

□

The following lemma provides a geometric bound on the distribution of $X_{[i]\to j}$ and $T_{[i]}$ variables, and will be useful to then bound their expectations.

Lemma 5.2. *Let $[i]$ be a node in the tree, and (if $[i]$ is not a leaf) $[[i],j]$ be an arbitrary child of $[i]$. There exist positive constants $A_{[[i],j]}, B_{[[i],j]}, C_{[i]}, D_{[i]}$ such that, for all $\alpha \in [0,1)$:*

$$\begin{aligned}\mathbf{P}[X_{[i]\to j} = x] &\leq A_{[[i],j]}(B_{[[i],j]}\alpha)^{x-1}, \forall x \geq 1\\ \mathbf{P}[T_{[i]} = k] &\leq C_{[i]}(D_{[i]}\alpha)^{k-1}, \forall k \geq 1\end{aligned}$$

Proof. by induction over the structure of the tree. If $[i]$ is a leaf, then $T_{[i]} = 1$ and the result holds with $C_{[i]} = 1, D_{[i]} = 1$. Otherwise, assume that $[i]$ is not a leaf, and that the result holds for all children of $[i]$. Let $[[i],j]$ be an arbitrary child of $[i]$, and $x \geq 1$. Using the induction hypothesis, and the fact that $\alpha \leq 1$:

$$\begin{aligned}\mathbf{P}[X_{[i]\to j} = x] &= \sum_{k=1}^{x} \mathbf{P}[T_{[[i],j]} = k]\binom{x-1}{k-1}\alpha^{x-k}(1-\alpha)^k\\ &\leq \sum_{k=1}^{x} C_{[[i],j]}(D_{[[i],j]}\alpha)^{k-1}\binom{x-1}{k-1}\alpha^{x-k}(1-\alpha)^k\\ &= C_{[[i],j]}\alpha^{x-1}\sum_{k=1}^{x}(D_{[[i],j]})^{k-1}\binom{x-1}{k-1}(1-\alpha)^k\\ &= C_{[[i],j]}\alpha^{x-1}(1-\alpha)[1+D_{[[i],j]}(1-\alpha)]^{x-1}\\ &\leq C_{[[i],j]}\alpha^{x-1}[1+D_{[[i],j]}]^{x-1}\end{aligned}$$

The result for $X_{[i]\to j}$ follows, with $A_{[[i],j]} = C_{[[i],j]}$ and $B_{[[i],j]} = 1 + D_{[[i],j]}$.
The result for $T_{[i]}$ remains to be proven. For $t \geq 1$:

$$\begin{aligned}\mathbf{P}[T_{[i]} = k] &= \prod_{[[i],j]\in c([i])} \mathbf{P}[X_{[i]\to j} \leq k] - \prod_{[[i],j]\in c([i])} \mathbf{P}[X_{[i]\to j} < k]\\ &= \prod_{[[i],j]\in c([i])} \big(\mathbf{P}[X_{[i]\to j} < k] + \mathbf{P}[X_{[i]\to j} = k]\big) - \prod_{[[i],j]\in c([i])} \mathbf{P}[X_{[i]\to j} < k]\end{aligned}$$

When developing the first product, a term $\prod_{[[i],j]\in c([i])} \mathbf{P}[X_{[i]\to j} < k]$ appears, which cancels out with the second product. Remaining terms in the first product are indexed with σ. These terms contain one or more factors of the form $\mathbf{P}[X_{[i]\to j} = k]$ where $[[i],j]$ is a child of $[i]$, and other factors of the form $\mathbf{P}[X_{[[i],j']} < k]$. Let $j(\sigma)$ be one of the j such that $\mathbf{P}[X_{[[i],j(\sigma)]} = k]$ appears in the term. An upper-bound for the other factors is 1, effectively keeping only the contribution of $\mathbf{P}[X_{[[i],j(\sigma)]} = k]$:

$$\begin{aligned}\mathbf{P}[T_{[i]} = t] &\leq \sum_{\sigma} \mathbf{P}[X_{[[i],j(\sigma)]} = k] \times 1\\ &\leq \sum_{\sigma} A_{[[i],j(\sigma)]}(B_{[[i],j(\sigma)]}\alpha)^{k-1}\end{aligned}$$

The result for $T_{[i]}$ follows, with $C_{[i]} = \sum_\sigma A_{[[i],j(\sigma)]}$ and $D_{[i]} = \max_\sigma B_{[[i],j(\sigma)]}$. □

Lemma 5.3 provides an approximation of the expectations of $X_{[i]\to j}$ and $T_{[i]}$ variables, at order 1 in α, using lemmas 5.1 and 5.2.

Lemma 5.3. *Let $[i]$ be a node in the tree, and (if $[i]$ is not a leaf) $[[i],j]$ be an arbitrary child of $[i]$. The following approximations hold when $\alpha \to 0$:*

$$\begin{aligned}\mathbf{E}[X_{[i]\to j}] &= 1 + (1 + l([[i],j]))\alpha + \mathcal{O}(\alpha^2)\\ \mathbf{E}[T_{[i]}] &= 1 + l([i])\alpha + \mathcal{O}(\alpha^2)\end{aligned}$$

Proof. First, it will be shown that $\sum_{x=3}^{+\infty} x\mathbf{P}[X_{[i]\to j} = x] = \mathcal{O}(\alpha^2)$. By summing the inequalities in lemma 5.2, and provided that α is small enough ($\alpha < 1/B_{[[i],j]}$), it is possible to write:

$$\begin{aligned}\sum_{x=3}^{+\infty} x\mathbf{P}[X_{[i]\to j} = x] &\leq A_{[[i],j]}B_{[[i],j]}^2 \sum_{x=0}^{+\infty}(x+3)(B_{[[i],j]}\alpha)^x\alpha^2\\ &= A_{[[i],j]}B_{[[i],j]}^2 \frac{3 - 2B_{[[i],j]}\alpha}{(1 - B_{[[i],j]}\alpha)^2}\alpha^2\\ &= \mathcal{O}(\alpha^2)\end{aligned}$$

Then, using lemma 5.1, $\mathbf{E}[X_{[i]\to j}]$ can be approximated as:

$$\begin{aligned}\mathbf{E}[X_{[i]\to j}] &= \mathbf{P}[X_{[i]\to j} = 1] + 2\mathbf{P}[X_{[i]\to j} = 2] + \sum_{x=3}^{+\infty} x\mathbf{P}[X_{[i]\to j} = x]\\ &= 1 - (1 + l([[i],j]))\alpha + 2(1 + l([[i],j]))\alpha + \mathcal{O}(\alpha^2)\\ &= (1 + l([[i],j]))\alpha + \mathcal{O}(\alpha^2)\end{aligned}$$

which concludes the proof for $\mathbf{E}[X_{[i]\to j}]$. The proof for $\mathbf{E}[T_{[i]}]$ is similar. □

This allows proving theorem 5.1 for BIER reliability. In the following, $D([i])$ denotes the sum of depth of links in the tree rooted at $[i]$: $D([i]) = \sum_{l\in[i]} d(l)$.

Theorem (Traffic footprint for BIER)**.** *Let $[i]$ be a node in the tree. The following approximation holds when $\alpha \to 0$:*

$$\mathbf{E}[M_{[i]}^B] = l([i]) + D([i])\alpha + \mathcal{O}(\alpha^2)$$

Proof. by induction over the structure of the tree. The results holds for leaves, because $M_{[i]}^{\mathrm{B}} = l([i]) = D([i]) = 0$ by definition. Otherwise, let $[i]$ be a node that is not a leaf, and assume that the result holds for the children of $[i]$. Then:

$$\begin{aligned}\mathbf{E}[M_{[i]}^{\mathrm{B}}] &= \sum_{[[i],j]\in c([i])} \left[\mathbf{E}[X_{[i]\to j}] + \mathbf{E}[M_{[[i],j]}^{\mathrm{B}}]\right]\\ &= \sum_{[[i],j]\in c([i])} [1 + (1 + l([[i],j]))\alpha + l([[i],j]) + D([[i],j])\alpha + \mathcal{O}(\alpha^2)]\\ &= \left[\sum_{[[i],j]\in c([i])} 1 + l([[i],j])\right] + \left[\sum_{[[i],j]\in c([i])} 1 + (D([[i],j]) + l([[i],j]))\right]\alpha + \mathcal{O}(\alpha^2)\\ &= l([i]) + D([i])\alpha + \mathcal{O}(\alpha^2)\end{aligned}$$

In the last sum, the first term corresponds to the link from $[i]$ to a child $[[i],j]$, and the second term corresponds to the sum of depths of all links in the subtree rooted at $[[i],j]$ incremented by 1, *i.e.,* the depth as counted from the root $[i]$. □

A proof of theorem 5.1 for multicast reliability can now be expressed.

Theorem (Traffic footprint for multicast)**.** *The following approximation holds when $\alpha \to 0$:*

$$\mathbf{E}[M_{[1]}^m] = L + \left[L^2 - L(D-1)\right]\alpha + \mathcal{O}(\alpha^2)$$

Proof. Let $d(l)$ be the depth of a link l as seen by the root (a link between the root and one of its children having depth 1). Using equation (5.6), it is possible to write:

$$\begin{aligned}\mathbf{E}[F_{[1]}] &= \sum_l (1-\alpha)^{d(l)-1} \\ &= \sum_l \left[1-(d(l)-1)\alpha+\mathcal{O}(\alpha^2)\right] \\ &= L-L(D-1)\alpha+\mathcal{O}(\alpha^2)\end{aligned}$$

Combining proposition 5.2 and lemma 5.3 yields:

$$\begin{aligned}\mathbf{E}[M^{\mathrm{m}}_{[1]}] &= \mathbf{E}[F_{[1]}]\mathbf{E}[T_{[1]}] \\ &= (L-L(D-1)\alpha+\mathcal{O}(\alpha^2))(1+L\alpha+\mathcal{O}(\alpha^2)) \\ &= L+[L^2-L(D-1)]\alpha+\mathcal{O}(\alpha^2)\end{aligned}$$

which concludes the proof. □

Finally, the following proves theorem 5.1 for unicast reliability.

Theorem (Traffic footprint for unicast)**.** *The following approximation holds when* $\alpha \to 0$*:*

$$\mathbf{E}[M^{u}_{[1]}] = L + [C\Delta - L(D-1)]\,\alpha + \mathcal{O}(\alpha^2)$$

Proof. As in the proof for multicast reliability, the first term in equation (5.11) can be approximated as: $\mathbf{E}[F_{[1]}] = L - L(D-1)\alpha + \mathcal{O}(\alpha^2)$. Hence, the whole expectation can be approximated as:

$$\begin{aligned}\mathbf{E}[M^{\mathrm{u}}_{[1]}] &= \mathbf{E}[F_{[1]}] + \sum_{[c]\in\mathcal{C}} \frac{(1-(1-\alpha)^{d([c])})^2}{\alpha(1-\alpha)^{d([c])}} \\ &= \mathbf{E}[F_{[1]}] + \sum_{[c]\in\mathcal{C}} \frac{\alpha^2 d([c])^2 + \mathcal{O}(\alpha^3)}{\alpha} \\ &= L - L(D-1)\alpha + \mathcal{O}(\alpha^2) + \sum_{[c]\in\mathcal{C}} d([c])^2\alpha + \mathcal{O}(\alpha^2) \\ &= L + [C\Delta - L(D-1)] + \mathcal{O}(\alpha^2)\end{aligned}$$

concluding the proof. □

5.6 Summary of Results

This chapter has proposed a scalable network service offering efficient and reliable multicast. NACK-based, this network service uses BIER (Bit-Indexed Explicit Replication) for ensuring that traffic (both original transmissions and retransmissions) are forwarded over a minimal shortest path tree, requiring maintenance of neither per-flow nor per-group state by intermediate routers: the source will encode, for each (re)transmission of a multicast data packet, the precise destination set – be that every member of a given group, or those members having issued a NACK to request retransmission.

The performance of this network service is compared with "classic" reliable multicast mechanisms, where retransmissions are either unicast (to all destinations having sent a NACK) or multicast to all destinations in a given group (when a NACK for a multicast data packet was received from any destination).

Simulation studies in both data-center-like and in Internet-like topologies, and when faced with different loss models, show that the proposed BIER-based reliable multicast network service is able to achieve reliability, while overcoming the two main shortcomings of these reference mechanisms:

(i) contrary to multicast reliability, links not concerned by losses are not affected by retransmissions and (ii) contrary to unicast reliability, links concerned by losses do not unnecessarily carry multiple copies of the same packets.

Generalizing from the simulation studies, an analytical model is presented, which quantifies the retransmission footprint incurred by the three mechanisms in *any* topology – and which shows that the BIER-based reliable multicast network service incurs a consistently lower overhead.

Results from this chapter have been published in [81].

Chapter 6

Reliable BIER with Peer Caching

As the size and complexity of Data-Center Networks (DCNs) [151] and Content Distribution Networks (CDNs) [152] grow, efficient multicast distribution of content becomes increasingly desirable [140,153,154]. Several protocols have been proposed, which offer reliable multicast delivery services. NORM (Negative-acknowledgement-Oriented Reliable Multicast) [128], among others, uses sequence numbers in data packets to detect packet losses, and negative acknowledgements (NACKs) which trigger a multicast transmission of the missing packet to the multicast group. Other protocols [129,140] use unicast retransmissions to those destinations having missed a packet.

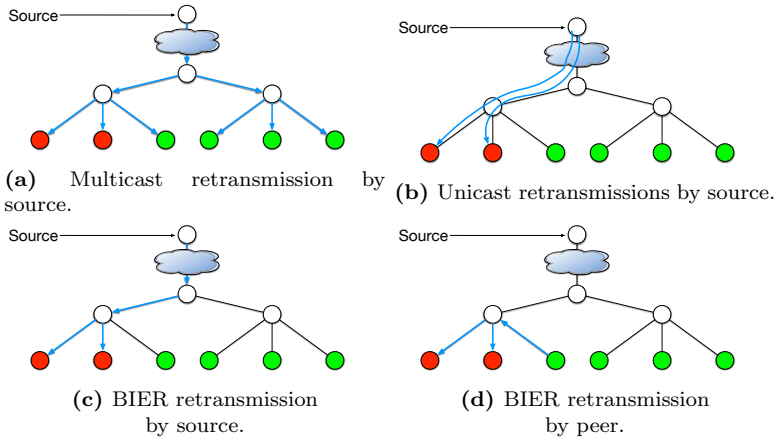
As introduced in section 1.2.2, BIER is a multicast protocol which removes the need for flow-state in intermediate routers by allowing compact, explicit, source-based specification of the set of destinations for each multicast packet. In chapter 5, BIER has been extended to offer reliable multicast delivery of content: using sequence numbers and NACKs for detecting and requesting retransmissions of missing packets, retransmissions themselves are made, using BIER, to the *exact* set of destinations having sent a NACK. This both avoids flooding the whole original multicast tree (figure 6.1a), and prevents duplicate retransmissions over the same link (as would be the case, for unicast retransmissions to each destination, figure 6.1b).

As with NORM, retransmissions in *reliable BIER* are done by the source (figure 6.1c). However, "local recovery", where retransmissions are performed by neighbors of destinations having missed a packet (figure 6.1d), may be preferable to retransmissions from the source [136, 140, 155]. For instance, this might be desirable for Quality-of-Experience reasons (*e.g.,* when the source and the destination having lost a packet experience a substantial round-trip delay), or when there are links towards the source whose usage should be limited (*e.g.,* so as to not overload the source, or for traffic engineering or economic reasons).

6.1 Statement of Purpose

The purpose of this chapter is to extend the *reliable BIER* mechanism introduced in chapter 5 to allow a destination having lost a multicast packet to request local retransmission thereof from local *peers* (*i.e.,* which are topologically close, and part of the destination set for the multicast flow) which may have successfully received a copy thereof – before requesting a retransmission from the source. This is achieved by (i) each destination caching successfully received packets for a small amount of time, and (ii) destinations detecting a packet loss sending a NACK through an ordered set of peers (candidates for retransmission) followed by the source, using Segment Routing.

The advantages of this approach are threefold: (i) the use of peer-based recovery reduces the number of retransmissions from the source, (ii) the use of SR allows a destination to generate a single NACK for a lost packet, which ultimately and automatically will be forwarded to the source if no local retransmissions are possible, and (iii) BIER-based retransmissions (from the source or from any peer) reduce the overall traffic by avoiding both unnecessary duplicate unicast retransmissions across a link close to the source, and multicast floods across the entire multicast tree.



(a) Multicast retransmission by source.

(b) Unicast retransmissions by source.

(c) BIER retransmission by source.

(d) BIER retransmission by peer.

Figure 6.1 – Comparison of different reliability mechanisms. In this example, red clients are assumed to have missed reception of a packet and sent a NACK. With "standard" multicast retransmissions [128], the source re-floods the whole tree as a result (a). With unicast retransmissions [129], the source will send multiple copies of the same packet over the same link (b). With BIER retransmissions [81], the source re-floods only the subset of failing destinations (c). With peer-based BIER retransmissions introduced in this chapter, the NACKs are sent to a peer that had cached the packet, which then floods the failing destinations, spanning a smaller tree (d).

6.1.1 Related Work

Work related to BIER and Segment Routing has already been reviewed in section 1.2.1 and in section 1.2.2. Different approaches to reliable multicast exist, most of which are *not* based on BIER, and have been reviewed in section 5.1.1.

Among those approaches, it is noteworthy that some of them use some form of caching, in intermediary nodes or in destinations. In LBRM [134], a specific node (the *log server*) is designated to cache packets from the source. With TMTP [136], destinations are grouped into different domains, in each of which local recovery can be performed. With RMTP [129], intermediate routers (*designated receivers*) can participate in retransmissions; similarly, PGM [139] allows recovery from *designated local repairers*. Finally, RDCM [140] uses unicast retransmissions in a backup overlay in which each peer is responsible for recoveries to at most two destinations.

The differences between these approaches and the mechanism introduced in this chapter are threefold: (i) routers are not involved in the caching mechanism, leaving this only to the destinations and therefore not requiring expanding the memory capability of the routing hardware; (ii) retransmissions conducted by peers use BIER and therefore will exclusively target those destinations having missed a packet; and (iii) the use of SR to probe peers for availability of cached packets allows for flexible and pluggable probing policies.

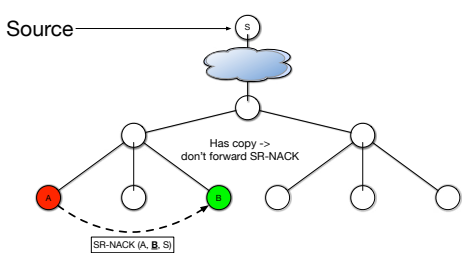
Information-Centric Networking (ICN)

The proposal in this chapter is built on edge-caching and cooperative content management, which has some notions in common with Information Centric Networking (ICN). ICN [22] is a paradigm where content is stored as named *data packets*, and where users send *interest packets* requesting named data packets. As data packets are only identified by names, caching by intermediate routers is possible, and a packet needs only be delivered once per interface, corresponding to a previously-emitted interest. ICN thus shares properties with reliable multicast protocols, by enabling recovery of data from nearby routers.

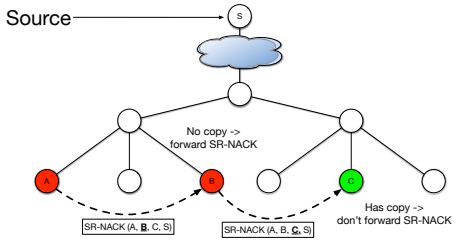
What is proposed in this chapter differs from ICN in that it (i) assumes push-based multicast applications and (ii) does not rely on routers performing caching, by offloading this task exclusively to the set of destinations – *i.e.,* requires no extensive modifications to routers nor to applications.

6.1.2 Chapter Outline

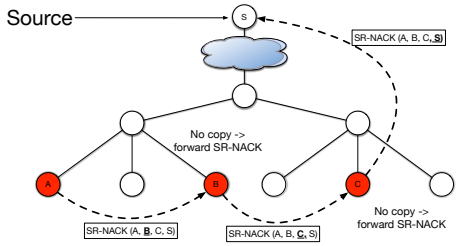
The remainder of this chapter is organized as follows. Section 6.2 gives a birds-eye view of the proposed extension to *reliable BIER*, and section 6.3 a detailed specification of how BIER is used



(a) SR-NACK to peer+source, peer retransmission.

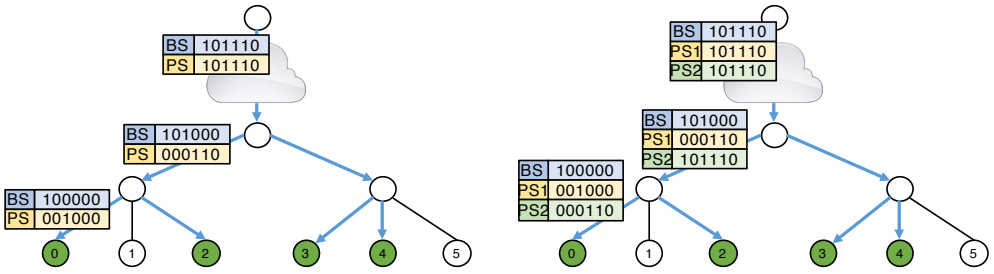


(b) SR-NACK to two peers+source, second peer retransmission.



(c) SR-NACK to two peers+source, source retransmission.

Figure 6.2 – SR-based recovery scenarios: SR-NACK sent by A. A red destination indicates that it is not able to satisfy the retransmission request, whereas a green destination indicates that it is.



(a) Single peerstring **(b)** Double peerstring

Figure 6.3 – Example of single- and double-peerstring operation

to construct a reliable multicast framework which can accommodate diverse policies for selecting peers for local retransmissions. Section 6.4 introduces basic taxonomy and example peer selection policies. Section 6.5 provides a theoretical analysis and discussion of performance and cost trade-offs for each of these policies, which are experimentally evaluated by way of network simulations in sections 6.6-6.8: section 6.6 introduces the characteristics of the simulation environment, and sections 6.7 and 6.8 describe and discuss the main performance results over both a data center topology and a real ISP topology. Finally, section 6.9 concludes this chapter.

6.2 Overview: Reliable BIER with Peer Caching

In this chapter, the *reliable BIER* mechanism introduced in chapter 5 is extended to support recovery from *peers*. Rather than sending a NACK directly to the source to request retransmission of a lost packet, this chapter proposes that a NACK be first sent through an ordered set of peer(s), each of which might be able to provide a retransmission if they have a cached copy of the lost packet (figure 6.1d). Retransmission from the source is solicited as a "last resort". As with *reliable BIER*, peers can aggregate NACKs before performing a BIER-based retransmission. This locality in retransmissions is expected to reduce delays, as well as to reduce the load on the source, and on its egress links [131, 155].

6.2.1 Segment Routing Recovery

Requesting retransmission from an ordered list of peers, followed by the source, is done by sending a NACK using Segment Routing (an "SR-NACK"). Each segment will trigger an action which is: (i) if the peer is unable to perform the retransmission, forward the packet to the next segment; (ii) if the peer is able to perform the retransmission, stop forwarding the segment and perform a BIER retransmission (figure 6.1d).

This is illustrated in figure 6.2: in (a) an SR-NACK is received by a peer which is able to satisfy the retransmission request; in (b) an SR-NACK is received by a peer which is not able to satisfy the retransmission request, and the next segment is another peer which, then, is able to satisfy the retransmission request; in (c) neither of the two peers receiving the SR-NACK is able to satisfy the retransmission request, and the SR-NACK is therefore forwarded to the source of the multicast packet.

The combined approach thus consists of the source performing an initial BIER transmission of a multicast packet. Destinations receiving the packet may (in addition to processing it) cache it for a short amount of time for possible peer-retransmission. A destination detecting a packet loss (*e.g.,* by receiving a subsequent packet belonging to the same multicast flow) will construct an SR-NACK, containing a number of peers followed by the source. A peer receiving an SR-NACK for which it is able to offer a retransmission will behave as if it was the source in *reliable BIER*: collect NACKs for this packet for a small amount of time, record the destinations requesting retransmission, and use BIER for retransmitting the packet to exactly the set of destinations from which an SR-NACK was received, when the timer expires.

6.2.2 Peer Caching with Peerstrings

This recovery mechanism is agnostic to the manner in which the set of candidate peers is chosen. If the network operator has instrumented the network in such a way that some peers are "better" candidates for retransmissions (*e.g.,* they are more likely to have cached packets, they are behind less costly or less lossy links, *etc.*), destinations can be administratively configured to send NACKs to those – with the drawback of requiring instrumentation and configuration. Thus, this chapter introduces a modification to the BIER forwarding plane, allowing destinations to learn about candidate peers.

To this end, an additional bitstring is introduced in BIER headers, henceforth denoted a *peerstring*. This peerstring is set so as to allow a destination, detecting a packet loss, to identify potential peers from which a retransmission can be requested. *A minima*, the peerstring is empty, which defaults to requesting retransmission from the source, as in *reliable BIER*. *A maxima*, the peerstring contains all destinations (*i.e.,* is a copy of the bitstring as inserted by the source) — and any peerstring in between these two extremes is valid.

With the goal of encouraging locality in retransmissions, one simple policy is that, for a given destination, the peerstring contains the set of destinations that share the same parent as itself. This is illustrated in figure 6.3a: when destination 0 receives a packet, the peerstring has a bit set for all destinations which have the same parent as itself (*i.e.,* destination 2). When a router forwards a *reliable BIER* packet over an interface i, it must, in addition to updating the bitstring for that interface, update the peerstring – essentially setting the peerstring to the union of the bitstrings for all other outgoing interfaces.

An extension to this principle is to include two peerstrings in a data packet received by a destination d: one peerstring indicating destinations with the same parent as d, and a second peerstring indicating destinations with the same grandparent (but not the same parent) as d – as illustrated in figure 6.3b. Thus, this affords more flexibility, but at the expense of more per-packet overhead. This mode of operation will be referred to as the *two-peerstrings mode*.

6.3 Specification

In the proposed BIER extension, the BIER header defined in [68] is extended to include an additional bitstring, the *peerstring*, denoted $PS1$ and described in section 6.2.2. When BIER operates in *two-peerstrings mode*, the header will also include a second peerstring, $PS2$. This header is included in all multicast data packets, and is processed by each intermediate router. Each multicast data packet also includes a reliable BIER *header*, defined in chapter 5, which

conveys flow identifiers and sequence numbers, and which allows detecting lost packets in a flow. This header only carries end-to-end semantics, and is not processed by intermediate BIER routers.

6.3.1 Source Operation

Packet transmission: The source adds a reliable BIER header to each multicast data packet, containing a flow identifier and a sequence number, as well as a standard BIER header [68] extended with peerstrings, as described above. The bitstring BS in the BIER header contains the set of destinations receiving packets from within the flow. The peerstring $PS1$ is set to BS and, if included, $PS2$ is also set to BS, as illustrated in figures 6.3a and 6.3b. The multicast source also caches a copy of each sent packet during a time interval Δt^s_{cache}.

Packet retransmission: When receiving an SR-NACK for a given packet, a source starts a timer Δt^s_{agg}, during which it collects potential subsequent SR-NACKs for the same packet from other destinations. Upon expiration of this timer, a retransmission of the packet is performed, with the set of destinations that have sent a NACK as the BIER bitstring.

6.3.2 Intermediate Router Operation

BIER bitstring processing: BIER packets are processed according to the BIER specification [8]. Only bits corresponding to destinations for which the shortest-path is via interface i are preserved in the bitstring contained in multicast data packets transmitted over that interface i.

Peerstring processing: Upon receipt of a *reliable BIER* packet with a bitstring BS^{in}, and before forwarding it over interface i (with outgoing bitstring BS^{out}_i), a router must update $PS1$ and, if included, also $PS2$. In *two-peerstrings* mode, first $PS2^{out}_i$ is set to $PS1^{in}$. Then, the peerstring $PS1^{out}$ is set to the OR of the bitstrings sent over all other interfaces (formally, $PS1^{out}_i \leftarrow \cup_{j \neq i} BS^{out}_j$)[1]. This way, the $PS1$ sent over each interface contains the set of those other destinations, to which this router has sent a copy of the packet. Note that the use of bitwise operators to compute peerstrings makes it a simple operation to be implemented in hardware.

6.3.3 Destination Operation

Packet reception: Upon receipt of a packet by a destination, the packet is cached for a duration of Δt^p_{cache} for potential retransmission to a peer. The peerstring(s) of the packet are inspected. The included $PS1$ is used for updating the set $\mathcal{P}_1$ of "local" peers. If included, $PS2$ is used for updating the set $\mathcal{P}_2$, of second-most local peers.

Packet loss: Upon detection of loss of a packet in a given flow (by receiving a packet in the same flow, whose *reliable BIER* header indicates a higher sequence number), a destination builds an SR-NACK packet. The SR-NACK contains a *reliable BIER* header with the flow identifier and the sequence number of the requested packet, and its segment list is set to $(p_1, \ldots, p_n, s)$, where the p_i's are peers selected from $\mathcal{P}_1$ and $\mathcal{P}_2$, and where s is the source. Different policies can be used for deciding which peers to include from $\mathcal{P}_1$ and $\mathcal{P}_2$, and in which order, as described in section 6.4.

While sending the SR-NACK, the destination starts a Δt^d_{retry} timer. When this timer expires, if no retransmission is received, the same SR-NACK is retransmitted – until either the missing packet is received, or a retransmission request limit R_{lim} is reached – after which recovery is aborted. (If semi-reliability is unacceptable, R^d_{lim} must be set to ∞.)

Packet retransmission: Upon receipt of an SR-NACK, a peer inspects the reliable BIER header of the SR-NACK and extracts the flow identifier and sequence number. If a cached copy of the requested packet is available, it is scheduled for retransmission; otherwise, the SR-NACK is forwarded to the next entry in the segment list (*i.e.,* to the next peer or, ultimately, to the source).

Retransmissions from peers use the same mechanism as those from the source: a timer Δt^p_{agg} is used to collect other NACKs before sending the copy to the set of destinations which have NACKed the packet.

[1] $PS1^{out}_i \leftarrow BS^{in} \setminus BS^{out}_i$ is an equivalent way of proceeding, as $\cup_{j \neq i} BS^{out}_j = BS^{in} \setminus BS^{out}_i$ holds as an invariant.

6.4 Peer Selection Policies

As introduced in section 6.3.3, upon missing a packet, a destination will build an SR-NACK with peer(s) extracted from $\mathcal{P}_1$ and $\mathcal{P}_2$. The framework introduced in this chapter is agnostic to the policy used for selecting those peers. To illustrate this, the remainder of this section suggests *examples* of *simple* policies for selection of peers to be included in SR-NACKs: (i) random selection of peers, (ii) clustered selection of peers, and (iii) a simple adaptive (statistically-driven) policy.

6.4.1 Random Peer Selection

This policy builds an SR segment list (p, s) where s is the source, and p is a peer randomly selected from $\mathcal{P}_1$. Randomly selecting peers from $\mathcal{P}_1$ may increase locality of retransmissions, but rarely allows aggregation of multiple retransmissions into a single BIER packet. As an example, if ten destinations $d_1, \ldots, d_{10}$ have the same parent (thus, for each d_i, $\mathcal{P}_1 = \{d_1, \ldots, d_{10}\} \setminus \{d_i\}$), and d_1, d_2 both detect loss of the same packet, the probability that d_1 and d_2 send an SR-NACK for this packet to the same peer in $\{d_3, \ldots, d_{10}\}$ is 1/8.

6.4.2 Deterministically Clustered Peer Selection

This policy builds an SR segment list (p, s) such that all destinations with the same parent router (*i.e.,* all destinations with the same $\mathcal{P}_1$) select the same p. As a convenient convention, for this chapter, all d will select as p the element in $\mathcal{P}_1 \setminus \{d\}$ with the highest index.

This policy generalizes for *two-peerstring mode* by building a SR segment list (p_1, p_2, s). All d will select as p_1 the element in $\mathcal{P}_1 \setminus \{d\}$ with the highest index, and as p_2 the element in $\mathcal{P}_2 \setminus \{d\}$ with the highest index.

While this policy favours aggregation of local retransmissions into a single BIER packet, it does not guard against selecting an unsuitable peer, *e.g.,* a peer located behind a particularly lossy link.

6.4.3 Adaptive Statistically-driven Peer Selection

As long as the constraint that the last segment in the SR segment list for a SR-NACK must be the source is satisfied, any adaptive policy – allowing a destination to observe and "learn" which peers are good candidates from whom to request retransmissions – can be be used for selecting additional peers for inclusion.

Formally, this is an instance of the Multi-Armed Bandit problem: an agent (a destination needing a retransmission) repeatedly activates one of several casino arms (in this case: sends an SR-NACK to a peer) and collects a reward (here: obtains a retransmission or not). The goal is a policy which maximizes the expected reward [156–158].

As an illustration, a simple ε-greedy peer selection policy is employed: with probability ε ($\varepsilon \ll 1$), a destination detecting a packet loss sends an SR-NACK to a random peer among the set of available peers. With probability $(1 - \varepsilon)$, it sends an SR-NACK to the peer from which it has received the highest number of successful retransmissions so far. The value of ε reflects the trade-off between *exploration* (contacting random peers and gathering statistics) and *exploitation* (requesting retransmission from the best known candidate) – between reactivity to changes and performance after convergence.

6.5 Policy Analysis

The impact of using peer caching and peer retransmissions, and of each of the policies introduced in section 6.4, can be quantified analytically. Section 6.5.1 provides a basic analysis of the benefits of local retransmissions, and section 6.5.2 derives an analytical model for the *random* and *clustered* policies. Section 6.5.3 then explores the benefits from a simple, adaptive policy.

6.5.1 Recovery Locality

The assumption in this chapter is that sending local recovery requests to a local peer is likely to be both "cheaper" than sending the request to the multicast source, and successful – *i.e.,* a "close" neighbour is likely to have successfully cached the requested packet. This section explores

the latter of these two assumptions, by quantifying the probability distribution of the distance from a destination having not received a given packet, to the closest peer that has (and is therefore able to perform a retransmission).

A regular tree topology is assumed, wherein inner nodes are intermediate BIER routers, and leaves are destinations. Nodes at a given depth are assumed to have the same number of children, and links at a given depth have the same loss probability. As corresponds to the operation of BIER, the root node of the tree is assumed to be the multicast source.

The tree is of height h, with node ranks indexed by their depth in the tree, from 0 (the source) to h. Similarly, links ranks are indexed from 0 (links from source to first descendants) to $h-1$. Each node at rank $i \in \{0,\dots,h\}$ has c_i children (with $c_h = 0$) and α_i is the loss probability of links at rank $i \in \{0,\dots,h-1\}$. Multicast transmission of a single packet to all leaves is considered.

Lemma 6.1 gives the probability that no nodes in a subtree rooted at a given node receives the multicast transmission.

Lemma 6.1. *The probability b_i that a multicast transmission from a node at rank $i \in \{0,\dots,h\}$ does not reach any destination (within its subtree), is:*

$$b_i = [\alpha_i + (1-\alpha_i)b_{i+1}]^{c_i} \tag{6.1}$$

Proposition 6.1. *The distribution of D, defined as the shortest distance from an arbitrary destination to a destination having successfully received the multicast transmission can for $k \in \{1,\dots,h\}$ be derived from* (6.1)*:*

$$\begin{aligned} f_D(2k) = &\left[\prod_{i=0}^{h-k-1}(1-\alpha_i)\right](\alpha_{h-k} + (1-\alpha_{h-k})b_{h-k+1}) \\ &\times [1 - [(1-\alpha_{h-k})b_{h-k+1} + \alpha_{h-k}]^{c_{h-k}-1}] \end{aligned} \tag{6.2}$$

with:

$$\begin{cases} f_D(0) &= \prod_{i=0}^{h-1}(1-\alpha_i) \\ f_D(\infty) &= b_0 \end{cases}$$

In proposition 6.1, $f_D(0)$ corresponds to the probability that a destination has successfully received the multicast transmission.

Let $\bar{D}$ denote the distance towards the closest successful destination, as seen from a destination having not received the packet. The probability distribution of $\bar{D}$ can be computed as the conditional distribution of D (see proposition 6.1) given that the packet is not received (which has probability $1 - f_D(0)$), as shown in corollary 6.1.

Corollary 6.1. *The distribution of $\bar{D}$, defined as the minimum distance from a destination that missed a packet to a destination which successfully received the packet is, for $k \in \{1,\dots,h\}$:*

$$\begin{aligned} f_{\bar{D}}(2k) &= \frac{f_D(2k)}{1-f_D(0)} \\ &= \frac{\prod_{i=0}^{h-k-1}(1-\alpha_i)}{1-\prod_{i=0}^{h-1}(1-\alpha_i)}(\alpha_{h-k} + (1-\alpha_{h-k})b_{h-k+1}) \\ &\quad \times [1 - [(1-\alpha_{h-k})b_{h-k+1} + \alpha_{h-k}]^{c_{h-k}-1}] \end{aligned} \tag{6.3}$$

with:

$$\begin{cases} f_{\bar{D}}(0) &= 0 \\ f_{\bar{D}}(\infty) &= \frac{b_0}{1-f_D(0)} = \frac{b_0}{1-\prod_{i=0}^{h-1}(1-\alpha_i)} \end{cases}$$

The probabilities α_i of loss at each link appear in equations (6.2) and (6.3), allowing to reflect different topology assumptions. In some topologies, loss probabilities may, *e.g.,* be assumed negligible for links close to the source, and more significant for links close to destinations (*i.e.,* $\alpha_i < \alpha_j$ if $i < j$). Two models for rank-dependent link loss probabilities are considered:

- Linear increase (`lin`): $\alpha_i = \alpha_{\max}\frac{i}{h-1}$.
- Exponential increase (`exp`): $\alpha_i = \alpha_{\max}\frac{e^i-1}{e^{h-1}-1}$.

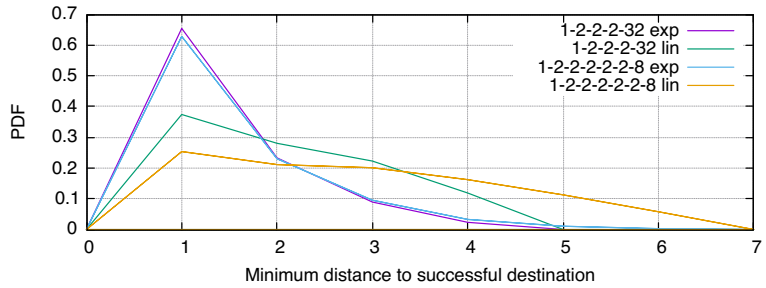


Figure 6.4 – PDF of the recovery at $2k$ hops (conditioning to a loss).

Figure 6.5 – Example with a policy X over $n = 10$ destinations, with $K = 4$ destinations (2, 5, 6 and 7) having lost a packet. Arrows indicate recovery requests (SR-NACKs): destination 2 requests recovery from peer 1; destination 5 from peer 2; and destinations 6 and 7 from peer 10; $S_X = 3$ recovery requests are successful (*i.e.,* those from 2, 6 and 7), as peers 1 and 10 have correctly received the original packet; $T_X = 2$ retransmissions are performed (from 1 and 10).

Figure 6.4 depicts the distribution of the minimum distance from a destination that missed a multicast transmission, and to a destination successfully receiving a multicast transmission, $f_{\bar{D}}(2k)$. Two tree topologies with 256 destinations are considered: (i) $h = 5$, $(c_0, \ldots, c_5) = (1, 2, 2, 2, 32)$; and (ii) $h = 7$, $(c_0, \ldots, c_7) = (1, 2, 2, 2, 2, 2, 8)$.

It can be observed that the expected shortest distance between a failing destination and a successful peer is lower for the exponential loss model (in which the loss probability for top links is lower) than for the linear loss model. In other words, as expected, if top links are less lossy, it is more likely for a failing destination to be able to recover from a close destination.

This confirms the intuition that in networks with this type of loss distribution (such as can be envisioned in data-centers, or in networks where the "last hop" is for instance a wireless link or a consumer grade residential xDSL), selection of local recovery peers (*e.g.,* peers that are in the same subtree or in the immediately upper subtree) should be preferred to selection of peers farther away.

6.5.2 Clustered and Random Policies

The properties and performance of the two simplest (and most "extreme") static peer policies of section 6.4 – random and clustered peer selection – is studied by way of two metrics: the number of *recovery successes* (performance) and the number of incurred *recovery retransmissions* (cost). A recovery is *successful* if a destination detecting the loss of a packet sends an SR-NACK to a peer that previously has received and cached a copy thereof. The number of *recovery retransmissions* incurring is the number of *unique* peers, which are selected for retransmission by the set of destinations which have lost that packet. This is because each selected peer will send only one BIER transmission as a response to receiving a set of SR-NACKs for the same multicast packet.

For the remainder of this section, a subtree with n destinations is considered, and recovery of one packet within this subtree is examined.

The following variables are introduced (where X denotes a particular selection policy: R for the random policy and D for the clustered policy):

- K, the number of destinations which did *not* receive the packet by way of the original multicast transmission from the source;
- S_X, the number of *recovery successes*, *i.e.,* destinations which did *not* receive the packet by way of the original multicast transmission from the source, but which successfully received a retransmission from a peer, in response to an SR-NACK.
- T_X, the number of *recovery retransmissions*, *i.e.,* of unique peers which received an SR-NACK for a given multicast packet.

Figure 6.5 shows an example with $n = 10$ destinations, $K = 4$ destinations missing a packet, $S_X = 3$ successful requests (out of 4) and $T_X = 2$ retransmissions.

Lemmas 6.2 and 6.3 describe the probability density function (PDF) of the number of recovery successes (S_R, S_D) and of the number of retransmissions (T_R, T_D), for the random and clustered policies, respectively. For the number of retransmissions with the clustered policy, it is assumed that a peer, from receiving the first SR-NACK and until retransmission of the packet, waits a sufficiently large amount time so as to maximize the ability of aggregating retransmissions into a single BIER-transmission.

Lemma 6.2 (Random selection policy)**.** *Given $K = k$ destinations ($0 \leq k \leq n$) that* did not *receive the multicast transmission from the source, the probability that, from among these k destinations, s ($0 \leq s \leq k$) will choose a peer which* did *receive the multicast transmission from the source, $\mathbf{P}[S_R = s|K = k] \equiv f_{S_R,k}(s)$, is, under the random selection policy:*

$$f_{S_R,k}(s) = \binom{k}{s} \left(\frac{k-1}{n-1}\right)^{k-s} \left(\frac{n-k}{n-1}\right)^{s} \tag{6.4}$$

Given s recovery successes, the probability of t retransmissions ($0 \leq t \leq s$), $\mathbf{P}[T_R = t|S_R = s, K = k] \equiv f_{T_R,k,s}(t)$, is:

$$f_{T_R,k,s}(t) = \binom{n-k}{t} \sum_{i=0}^{t} (-1)^{t-i} \binom{t}{i} \left(\frac{i}{n-k}\right)^{s} \tag{6.5}$$

Proof. Consider the transmission of a multicast packet to a cluster of n destinations, from among which k do not receive the packet. Each of the k non-successful destinations then selects a peer at random among $(n-1)$ peers (all peers but itself), and there are $(n-k)$ successful peers, thus this selection is successful with probability $\frac{n-k}{n-1}$. Therefore, the probability that s of the k non-successful destinations sends a NACK to a successful peer follows a binomial law with parameter $\frac{n-k}{n-1}$, yielding equation (6.4).

To derive equation (6.5), assume that there are s successful recoveries. The number of retransmissions t incurred by these s recoveries corresponds to the number of unique elements sampled when drawing with replacement s elements from a set of $(n-k)$ elements. From [159], this is:

$$\frac{1}{(n-k)^s} \times \frac{(n-k)!}{(n-k-t)!} \times \left\{ {s \atop t} \right\}$$

where $\left\{ {s \atop t} \right\}$ (Stirling number of second kind) represents the number of ways to partition s elements into t non-empty subsets, and can be expressed as:

$$\left\{ {s \atop t} \right\} = \frac{1}{t!} \sum_{i=0}^{t} (-1)^{t-i} \binom{t}{i} i^s$$

Combining these two expressions yields equation (6.5), which concludes the proof. □

Lemma 6.3 (Clustered selection policy)**.** *The probability that s destinations (out of $k \geq 2$ destinations that* did not *receive the multicast transmission from the source) will choose a peer which* did *receive the multicast transmission from the source, $\mathbf{P}[S_D = s|K = k] \equiv f_{S_D,k}(s)$, is, under the clustered selection policy:*

$$f_{S_D,k}(s) = \begin{cases} \frac{n-k}{n} & \text{if } s = k \\ \frac{k}{n}\left(1 - \frac{k-1}{n-1}\right) & \text{if } s = 1 \\ \frac{k}{n}\frac{k-1}{n-1} & \text{if } s = 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

The probability of having t retransmissions, $\mathbf{P}[T_D = t|K = k] \equiv f_{T_D,k}(t)$, is:

$$f_{T_D,k}(t) = \begin{cases} 1 - \frac{k}{n}\frac{k-1}{n-1} & \text{if } t = 1 \\ \frac{k}{n}\frac{k-1}{n-1} & \text{if } t = 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.7}$$

As edge cases, when $k = 1$, there is one recovery/retransmission: $S_D = T_D = 1$; and when $k = 0$, there are no recoveries/retransmissions: $S_D = T_D = 0$.

Proof. Consider the clustered policy, whereby a failing destination sends a NACK to a designated peer p, and p itself if failing sends NACKs to another designated peer p^*. Consider the transmission of a multicast packet to a cluster of n destinations, from among which k do not receive the packet. Each of the k non-successful destinations then sends a recovery request (NACK) to p (or p^* if the non-successful destination is p itself). If the designated peer p is not amongst the k failing destinations (which happens with probability $\frac{n-k}{n}$), all the recoveries are successful, and $S_D = k$. If the designated peer p is among the failing destinations, but not its retransmitter p^* (which happens with probability $\frac{k}{n}\left(1 - \frac{k-1}{n-1}\right)$), then only one recovery request is successful (the one from the designated peer p), and $S_D = 1$. Finally, if both the designated peer p and its retransmitter p^* are among the k failing destinations (which happens with probability $\frac{k}{n}\frac{k-1}{n-1}$), no retransmission request is successful, and $S_D = 0$. Combining these three cases yields equation (6.6).

To derive equation (6.7), it suffices to note that both cases $S_D = k$ and $S_D = 1$ yield one retransmission, and that $S_D = 0$ yields zero retransmissions. Thus $\mathbf{P}[T_D = 1|K = k] = \mathbf{P}[S_D = k|K = k] + \mathbf{P}[S_D = 1|K = k]$ and $\mathbf{P}[T_D = 0|K = k] = \mathbf{P}[S_D = 0|K = k]$, yielding equation (6.7). □

From lemmas 6.2 and 6.3 follow two key results, describing the average number of recovery successes (proposition 6.2) and the average number of retransmissions (proposition 6.3 and corollary 6.2).

Proposition 6.2. *Assuming $K = k$ destinations that* did not *receive the multicast transmission from the source, the expected number of recovery successes for the random and clustered approaches is the same, and has the value:*

$$\mathbf{E}[S_R|K = k] = \mathbf{E}[S_D|K = k] = \frac{k(n-k)}{n-1} \tag{6.8}$$

Proof. Assuming that k destinations have missed the multicast packet, S_R has a binomial distribution with k samples and with success probability $\frac{n-k}{n-1}$. Thus, $\mathbf{E}[S_R|K = k] = k \times \frac{n-k}{n-1}$ by definition, which proves equation (6.8) for the random policy.

From equation (6.6), the expected value of S_D can be computed as:

$$\begin{aligned}\mathbf{E}[S_D|K = k] &= k \times \frac{n-k}{n} + 1 \times \frac{k}{n}\left(1 - \frac{k-1}{n-1}\right) \\ &= \frac{k}{n-1}(n-k)\end{aligned}$$

which proves that $\mathbf{E}[S_R|K = k] = E[S_D|K = k]$. □

Proposition 6.3. *Assuming $K = k$ destinations that* did not *receive the multicast transmission from the source, the expected number of retransmissions for random (S_R) and clustered (S_D) policies have the following expressions:*

$$\mathbf{E}[T_R|K = k] = (n-k) - (n-k)\left(\frac{n-2}{n-1}\right)^k \tag{6.9}$$

$$\mathbf{E}[T_D|K = k] = \begin{cases} 1 - \frac{k(k-1)}{n(n-1)} & \text{if } k \geq 1 \\ 0 & \text{if } k = 0 \end{cases} \tag{6.10}$$

Proof. Assuming that k destinations have missed the multicast packet, and that s recoveries are successful, the expected number of incurred recovery retransmissions for the **random policy** is computed by averaging equation (6.5):

$$\begin{aligned}\mathbf{E}[T_R|S_R = s, K = k] &= \sum_{t=0}^{s} t \times \mathbf{P}[T_R = t|S_R = s, K = k] \\ &= (n-k)\left[1 - \left(\frac{n-k-1}{n-k}\right)^s\right]\end{aligned}$$

Then, the average number of retransmissions with no assumption on s can be obtained by combining this result with equation (6.4), yielding (with $p = \frac{k-1}{n-1}$ to ease notation):

$$\begin{aligned}\mathbf{E}[T_R|K = k] &= \sum_{s=0}^{k} \mathbf{E}[T_R|S_R = s, K = k] \times \mathbf{P}[S_R = s, K = k] \\ &= (n-k) - (n-k)\sum_{s=0}^{k}\binom{k}{s}p^{k-s}(1-p)^s\left(\frac{n-k-1}{n-k}\right)^s \\ &= (n-k) - (n-k)\left(\frac{n-2}{n-1}\right)^k\end{aligned}$$

For the **clustered policy**, deriving the expected number of retransmissions is done by averaging equation (6.7), and by noting that since the number of retransmissions is either 0 or 1, the expected value is simply the probability that there is one retransmission:

$$\begin{aligned}\mathbf{E}[T_D|K = k] &= \mathbf{P}[T_D = 1|K = k] \\ &= \begin{cases} 1 - \frac{k(k-1)}{n(n-1)} & k \geq 1 \\ 0 & k = 0 \end{cases}\end{aligned}$$

concluding the proof. □

The following corollary compares the behavior of both policies when considering that each destination has, independently, the same probability of having not received the multicast transmission from the source.

Corollary 6.2. *Assuming that each destination has (independently of the others) a probability $\beta \in [0, 1]$ of not having received the multicast transmission from the source (i.e., the probability of having k destinations which* did not *receive the multicast transmission from the source is binomial with parameter β, and thus $\mathbf{P}[K = k] = \binom{n}{k}\beta^k(1-\beta)^{n-k}$), then the expected number of retransmissions with the random policy grows linearly with the number of destinations, whereas the expected number of transmissions with the clustered policy is bounded:*

$$\begin{cases} \mathbf{E}[T_R] \sim_{n\to\infty} n(1-\beta)(1-e^{-\beta}) & = \Theta(n) \\ \mathbf{E}[T_D] \sim_{n\to\infty} 1-\beta^2 & = \Theta(1) \end{cases} \tag{6.11}$$

Proof. Assuming that each destination has missed the multicast packet independently with probability β, the number K of failing destinations follows a binomial distribution:

$$\mathbf{P}[K = k] = \binom{n}{k}\beta^k(1-\beta)^{n-k} \tag{6.12}$$

For the **random policy**, it is possible to deduce the expected number of retransmissions (averaged over the failing destinations k) by combining equations (6.9) and (6.12):

$$\begin{aligned}\mathbf{E}[T_R] &= \sum_{k=0}^{n} \mathbf{E}[T_R|K = k] \times \mathbf{P}[K = k] \\ &= (n - n\beta) - \sum_{k=0}^{n}\binom{n}{k}\beta^k(1-\beta)^{n-k}(n-k)\left(\frac{n-2}{n-1}\right)^k \\ &= n(1-\beta)\left[1 - \left(1 - \frac{\beta}{n-1}\right)^n\right]\end{aligned}$$

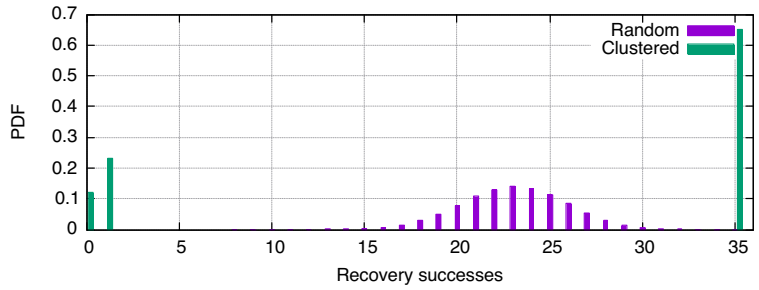


Figure 6.6 – Number of destinations able to obtain a retransmission of a packet from a peer, when $k = 35$ out of $n = 100$ *have not* received the initial multicast transmission.

which is, if $n \to \infty$ and $\beta \neq 0, \beta \neq 1$, using Taylor's 1st-term approximation of $\left(1 - \frac{\beta}{n-1}\right)^n$:

$$\mathbf{E}[T_R] = n(1-\beta)(1-e^{-\beta}) + \mathcal{O}(1)$$

The same reasoning can be used for the **clustered policy**. The expected number of retransmissions becomes, by using (6.10) and (6.12):

$$\begin{aligned}\mathbf{E}[T_D] &= \sum_{k=0}^{n} \mathbf{E}[T_D|K=k] \times \mathbf{P}[K=k] \\ &= \sum_{k=1}^{n} \binom{n}{k} \beta^k (1-\beta)^{n-k} \left(1 - \frac{k}{n}\frac{k-1}{n-1}\right) \\ &= 1 - (1-\beta)^n - \beta^2\end{aligned}$$

which is, if $n \to \infty$ and $\beta \neq 0$:

$$\mathbf{E}[T_D] = 1 - \beta^2 + o(1)$$

concluding the proof. □

From proposition 6.2, both policies yield the same performance *on average*, *i.e.,* the expectation of the number of recovery successes is the same for both policies. The clustered policy achieves the same reliability as the random policy by concentrating recovery requests on a single peer, which allows aggregation of retransmissions into a single BIER retransmission. Since aggregation occurs less often in the random policy, in terms of retransmission cost, the random policy is more expensive (linear in the number of destinations *vs* constant), as shown in corollary 6.2.

This difference is only possible because random and clustered policies achieve their (equal) average performance through different probability density distributions, as shown in figure 6.6. In this example, it is assumed that $k = 35$ destinations out of $n = 100$ have not received the initial multicast transmission. While with the random policy there is negligible chance that no less than 15 and no more than 30 destinations are able to obtain a retransmission from a peer, the clustered policy operates on an all-or-nothing fashion: all 35 destinations will obtain a retransmission from a peer with high probability – but recovery may be mostly unsuccessful with non-negligible probability ($\sim 12\%$ for 0 successes, 23% for only one success over 35).

In other terms, given a Service Level Agreement (SLA) commitment specifying a minimum fraction of destinations $(1-\delta)$ (with $\delta \ll 1$) being served without the need for source retransmission (either because destinations receive the packet in the first BIER transmission from the source or because the first peer recovery request is successful), there is a higher probability that systems are SLA-compliant when using the random policy, than systems using the clustered policy, for small tolerance values ($\delta > 0.05$ in figure 6.7) – and the gap between random and clustered policies is substantial for a non-negligible interval ($\delta \in [0.05, 0.25]$ in figure 6.7). The probability that a $(1-\delta)$ fraction of destinations is served without resorting to source retransmissions, under policy X, corresponds to function $g_{n,\beta,X}(\delta)$ described in equation (6.13), and is illustrated in figure 6.7

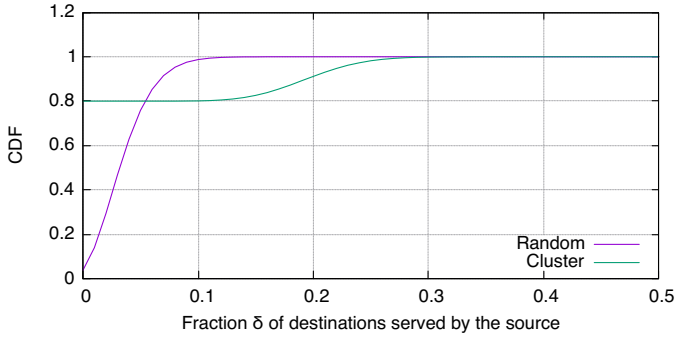
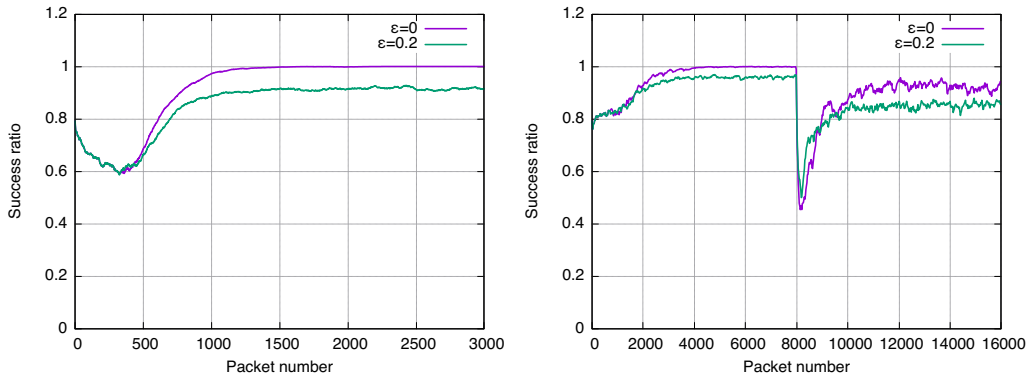


Figure 6.7 – Probability that a fraction $(1-\delta)$ of destinations successfully receive the packet directly from the source (in the first transmission) or from a contacted peer – *i.e.,* excluding source retransmissions, with $n = 100$, $\beta = 0.2$.



(a) Static scenario **(b)** Dynamic scenario

Figure 6.8 – Success ratio (after 1st recovery) for the ε-greedy policy, for static and dynamic scenarios.

for $n = 100$, $\beta = 0.2$, and random and clustered policies. It corresponds to:

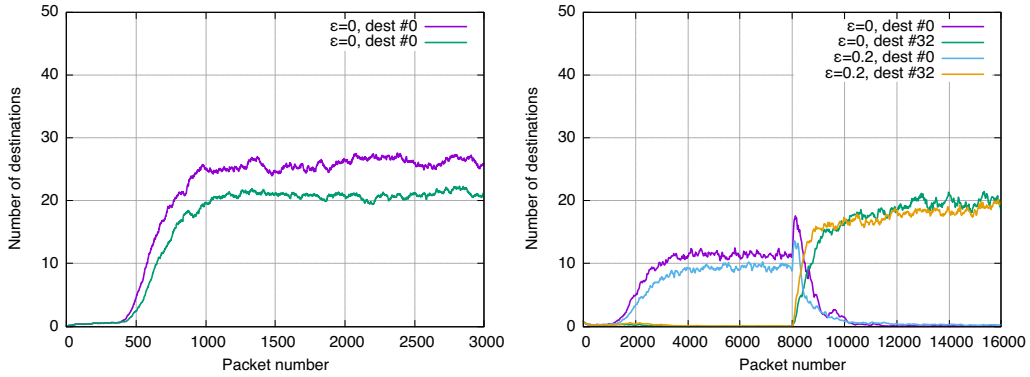
$$\begin{aligned} g_{n,\beta,X}(\delta) &= \mathbf{P}[S_X > K - n\delta] \\ &= \sum_{k=0}^{n} \mathbf{P}[S_X > K - n\delta | K = k]\mathbf{P}[K = k] \end{aligned} \tag{6.13}$$

where $\mathbf{P}[S_X > x | K = k]$ can be computed from equations (6.4) and (6.6). When source retransmissions are significantly less preferable (*e.g.,* due to a substantially higher delay incurred) than peer retransmissions, this may translate into a substantial quality degradation of the clustered policy with respect to the random policy. While the gap amplitude is dependent on the value of β (*i.e.,* lower loss probabilities at destination lead to shorter gaps, as it can be expected), the trend shown in figure 6.7 is invariant for values of n and β.

6.5.3 Going Adaptive: the ε-Greedy Policy

For the purpose of this analysis, a binary ($c = 2$) multicast BIER transmission tree with height $h = 7$ is considered, so as to illustrate the ability of the simple ε-greedy adaptive policy to learn and adapt to changes in networking conditions, under two different scenarios:

- A *static* scenario, where all links are lossy ($\alpha = 0.1$), except for links between the source and destination 0 – *i.e.,* with a static "best" (ideal) peer from which to request retransmission.
- A *dynamic* scenario which reflects a situation with failure of a "good" peer. Specifically destination 0 is, as in the static scenario and for the same reasons, the "best" (ideal) destination ($\alpha = 0$ for links from destination 0 and to the source) up until multicast packet number 8000, after which it becomes a "bad" destination behind highly lossy links ($\alpha = 0.4$).



(a) Static scenario **(b)** Dynamic scenario

Figure 6.9 – Number of destinations that *did not* receive the multicast transmission from the source, and which are selecting peers 0 and 32 under the ε-greedy policy, in static and dynamic scenarios

Concurrently, destination 32 is a relatively good, though not perfect, destination ($\alpha = 0.01$) during the entire duration of the flow.

Results of these simulations are reported in figure 6.8, which depicts the ratio of failing destinations choosing a successful peer as retransmission candidate, and figure 6.9, which depicts the number of failing destinations choosing destination 0 as a retransmission candidate[2]. It is worth observing that, in these simulations, recoveries are *idealized*: retransmissions from contacted peers are always successful if the contacted peer holds a copy of the requested BIER packet.

Unsurprisingly, $\varepsilon = 0$ (*i.e.,* choosing deterministically the destination with highest success record) achieves steadier performance than does $\varepsilon = 0.2$ (*i.e.,* choosing random destinations for recovery 20% of the time) in static conditions, as depicted in figure 6.8a. Using $\varepsilon = 0.2$, however, performance is less impacted by the failure of destination 0, and the system adapts faster to the new conditions, as depicted in figure 6.8b. As shown in figure 6.9b, failing destinations when using $\varepsilon = 0.2$ switch to destination 32 quicker, after the failure of former ideal destination 0.

6.6 Simulation Environment

The reliable multicast mechanism described in this chapterhas been implemented in `ns3` [147] as four components: (i) a BIER forwarding plane (as described in section 6.3.2), (ii) a Segment Routing forwarding plane, (iii) a reliable BIER layer for a source (section 6.3.1), and (iv) a reliable BIER layer for destinations (section 6.3.3).

The *reliable BIER* layer at the source interfaces with the UDP socket API, to transform UDP multicast packets into BIER packets, while also caching a copy of sent packets so as to be able to retransmit them on receipt of SR-NACKs. The *reliable BIER* layer at a destination also interfaces with the UDP socket API, collecting received BIER packets before handing them (in-order) over to the UDP socket. This layer also caches a copy of received packets, so as to be able to retransmit them on receipt of a SR-NACK from a peer, and generate SR-NACKs when a packet loss is detected.

The parameters as defined in section 6.3 are: $\Delta t^s_{agg} = \Delta t^p_{agg} = 7\ ms$, $\Delta t^p_{cache} = 50\ ms$, $\Delta t^s_{cache} = 100\ ms$, $\Delta t^d_{retry} = 15\ ms$, $R^d_{lim} = 3$. Notably, the value of the SR-NACK retry delay is set significantly greater than the NACK aggregation delay, $\Delta t^d_{retry} \gg \Delta t_{agg}$, so that a second NACK is only sent if the source (or peer) has had the chance to send a retransmission and has failed.

6.6.1 Links and Link Loss Model

All links are point-to-point, with 1 Gbps throughput and 1 μs propagation delay, have an MTU of 1500 bytes, and are attached to interfaces with drop-tail queues of size 512 packets.

[2]Results have been smoothed with an Exponential Window Moving Average filter with parameter $\alpha = 0.01$.

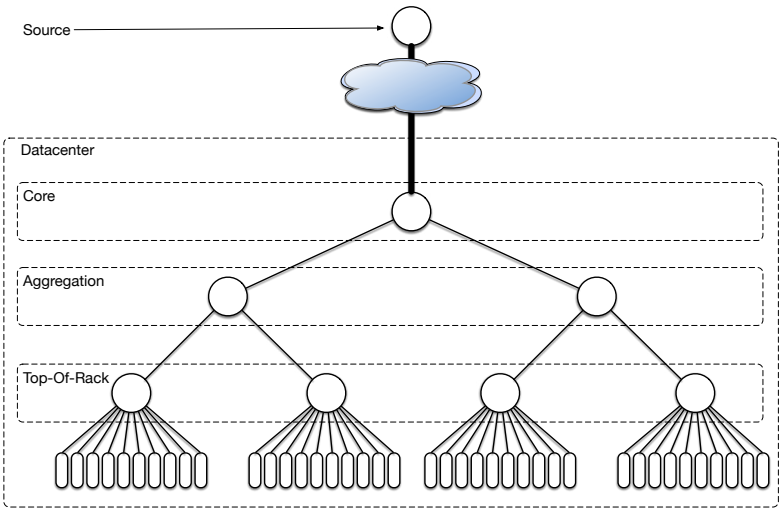


Figure 6.10 – Datacenter topology for simulations of section 6.7

The link loss model used for all links in the simulations is a clock-based Gilbert-Elliott model [148, 160], where the probability of a successful transmission is $k = 1$ in *good* state and $h = 0.5$ in *bad* state. Transitions from *bad* to *good*, and from *good* to *bad*, are triggered with exponential clocks, of mean $1/r = 2.5\ ms$ and $1/p = \frac{(h-\alpha)}{\alpha r} = \frac{(0.5-\alpha)}{\alpha} \times 2.5\ ms$, respectively, where $\alpha \in [0, 1]$ is a parameter representing "packet loss probability".

According to [149], the probability π_B of being in *bad* state is $\pi_B = \frac{p}{p+r} = \frac{1}{1+(h-\alpha)/\alpha} = \frac{\alpha}{0.5}$, yielding an expected link loss rate of $\pi_B(1-h) + (1-\pi_B)(1-k) = 0.5\pi_B = \alpha$. This justifies using α as a parameter to tune the average packet loss probability. This loss model is only applied to multicast data packets – SR-NACKs are not subject to losses, as the path from destinations to the source is supposedly less lossy[3].

6.7 Data-Center Simulations

For the purpose of evaluating the mechanism introduced in this chapter, a data-center-like topology is used. This reflects, *e.g.,* distributed storage, distribution of software upgrades, pre-placement of content, *etc.*

6.7.1 Network Topology

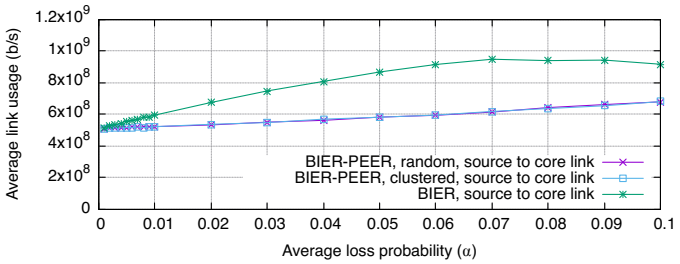
The topology used in this set of simulations is as follows, illustrated in figure 6.10:

- a source, "outside" the data-center – reachable across a capacity constrained connection (*e.g.,* a connection incurring higher delays, which has limited throughput, or subject to a higher, congestion-induced, loss probability) is attached to a core router;
- the core router is attached to 2 aggregation routers;
- each aggregation router is attached to 2 Top-of-Rack (ToR) routers;
- each ToR router is attached to 10 machines in a rack (destinations).

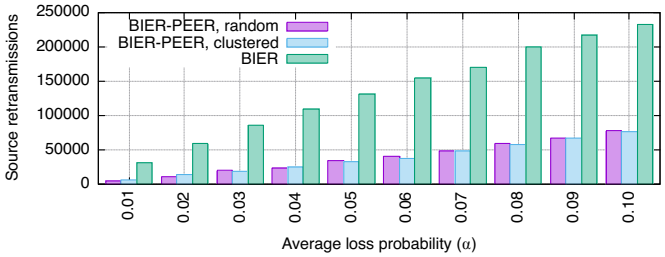
6.7.2 Evaluation Objectives

The objective of using SR-NACKs for requesting retransmissions from peers, rather than directly from the source, is specifically *not* to minimize the number of global transmissions – but rather to maximize the number of retransmissions that can be satisfied locally, *i.e.,* within the data-center. Consequently, one key metric is the number of retransmissions performed by the

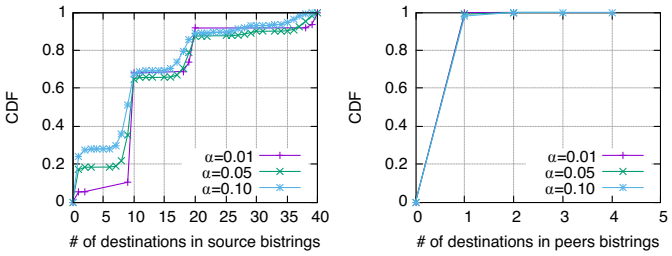
[3]Whereas traffic from the source to the destination is bursty, and therefore likely to cause interface buffers to run full and drop packets, traffic from destinations to the source is expected to be sparse (essentially, SR-NACKs).



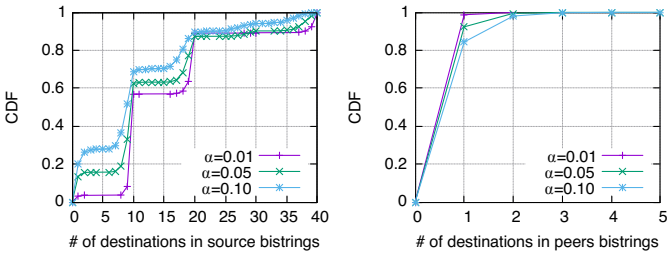
(a) Source-to-core link usage



(b) Number of retransmissions by the source



(c) Random peer selection: aggregation from source and peers



(d) Clustered peer selection: aggregation from source and peers

Figure 6.11 – Data-center topology: clustered vs random peer selection within same subtree, and for different values of the Gilbert-Elliott link loss probability α.

source – corresponding to the load of the link between the source and the core router (bold link in figure 6.10).

6.7.3 Static Peer Policy - One-Peerstring Mode

To baseline the benefits of locality, the two static peer selection policies introduced in section 6.4.1 (random) and in section 6.4.2 (clustered) are tested in one-peerstring mode: a destination, which detects that a packet has been lost, sends an SR-NACK towards first a "local" peer[4] according to the policy, then to the source. These are compared with simple *reliable BIER* without peer recovery (*i.e.,* wherein NACKs are sent directly to the source).

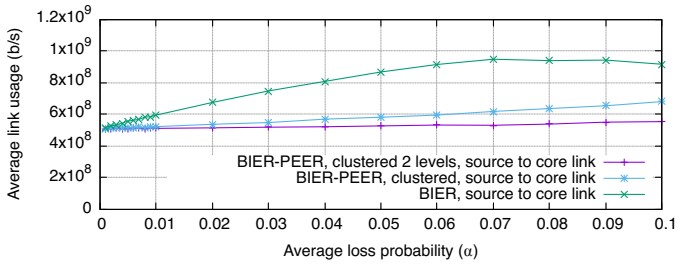
Figure 6.11 depicts the results of 19 four-second long simulations using a 500 Mbps multicast flow (*i.e.,* 166673 BIER-packets generated by the source), for different values of α (see section 6.6) applying uniformly to all links. Figure 6.11a shows the link usage of the ingress link to the core router (averaged over the duration of the simulation). Using simple *reliable BIER*, the source performs all retransmissions, and this link saturates for $\alpha \geq 7\%$. With peer-based retransmissions, even with $\alpha = 10\%$, this link remains well below saturation, with a link usage below 680 Mbps. This is detailed in figure 6.11a.

On receiving an SR-NACK, a peer waits for Δt^p_{agg}, to allow receiving SR-NACKs from other peers, before a single aggregate BIER retransmission is made. Thus, the number of destinations in the bitstring is an indicator of the ability to reduce the number of retransmissions. This is depicted in figures 6.11c and 6.11d, for random and clustered peer selection, and for retransmissions made by the source or by a peer.

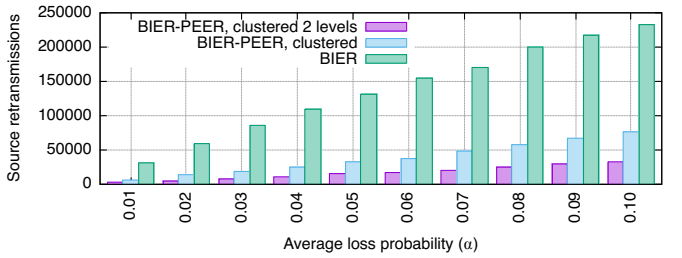
Retransmissions by the source are required when both the originator of an SR-NACK and the selected peer have not received a given packet – *e.g.,* when the packet was lost over the link between an aggregation router and a ToR router (see figure 6.10), in which case all the destinations below that ToR router would need to receive a retransmission, explaining the "step" at 10 destinations in source bitstrings (figures 6.11c and 6.11d). The similar step around 20 destinations in the source bitstrings is due to either a loss over the link between an aggregation router and the core router, or a packet being lost over two distinct links (between any aggregation router and ToR router).

Figures 6.11c and 6.11d also show that random peer selection (expectedly) does not facilitate aggregation, whereas clustered peer selection allows for some – since peers are only selected within the same subtree as that of the originator of the SR-NACK. Thus (i) a peer retransmission bitstring has ≤ 9 destinations, and (ii) if a loss is over a link between the ToR router and an aggregation

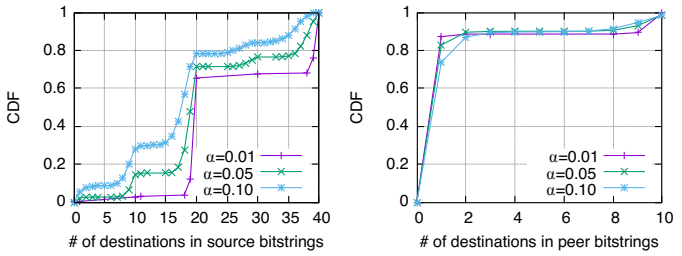
[4] A peer is considered "local" if it has the same parent – in which case, it is indicated in the received $PS1^{in}$.



(a) Source-to-core link usage



(b) Number of retransmissions by the source



(c) Clustered selection in two subtrees: aggregation from source and peers

Figure 6.12 – Data-center topology: clustered peer selection in two subtrees vs in one subtree, for different values of the Gilbert-Elliott link loss probability α.

router, only retransmissions from the source are possible.

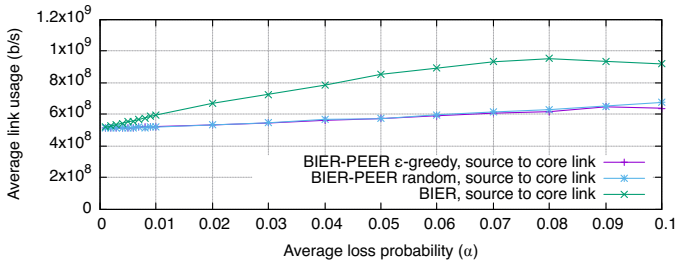
6.7.4 Static Peer Policy - Two-Peerstrings Mode

A variation of the clustered peer policy (section 6.4.2) is possible when using two peerstrings. A destination detecting that a packet has not been received will send an SR-NACK towards first the leftmost "local" peer, then the leftmost second-most local[5] peer, and finally the source. The objective is, again, to reduce the number of retransmissions needed from the source, at the cost of potentially more total (but local) traffic.

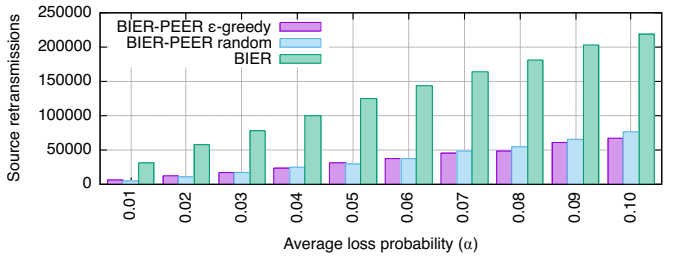
The baseline for this approach is the *clustered peer selection* policy of section 6.7.3, as well as standard *reliable BIER*, therefore the same topology, traffic patterns, and link loss model are used. The simulation results for different values of α are depicted in figure 6.12, where figure 6.12a shows that the two-peerstring approach yields a further reduction in the link usage on the ingress link to the core router, due to fewer retransmissions being required by the source. In figure 6.12b, this reduction appears to be from 2.2× less (when $\alpha = 0.05$) to 3.3× less (when $\alpha = 0.02$), when comparing to the one-peerstring approach.

Using the two-peerstring mode, retransmissions by the source are required only when neither the originator of an SR-NACK, nor its selected local peer, nor its selected second-most local peer, have received a given packet – *e.g.,* when the packet was either (i) lost over the link between an aggregation router and the core router, or (ii) lost over all of the links from an aggregation router and to the connected ToR routers (figure 6.10). In this case all the destinations below that aggregation router would need to receive a retransmission, which explains the "step" at

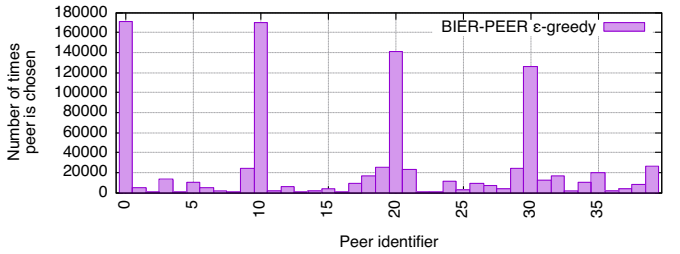
[5] A peer is considered "second-most local" if it has the same grandparent, but not the same parent – in which case, it is indicated in the received PS_2^{in}.



(a) Source-to-core link usage



(b) Number of retransmissions by the source



(c) Adaptive (ε-greedy) strategy in subtree: histogram of peer choices when in non-exploratory mode ($\alpha = 0.10$).

Figure 6.13 – Data-center topology with one ideal peer per rack: ε-greedy peer selection in subtree vs random, for different values of the Gilbert-Elliott link loss probability α.

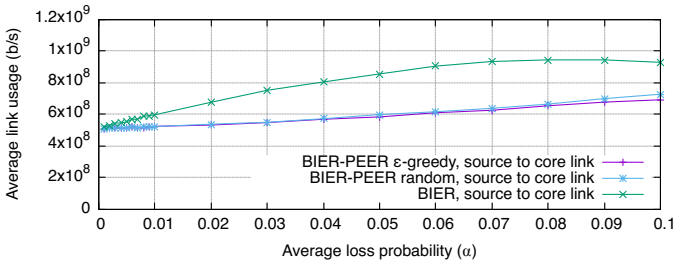
20 destinations in source bitstrings in figure 6.12c. The similar, but smaller, step around 10 destinations in the source bitstrings indicates the loss of a packet over only one of the links between an aggregation router and a ToR router. This confirms a greater degree of aggregation (among 20 destinations, rather than 10) of source retransmissions.

6.7.5 Adaptive Peer Policy

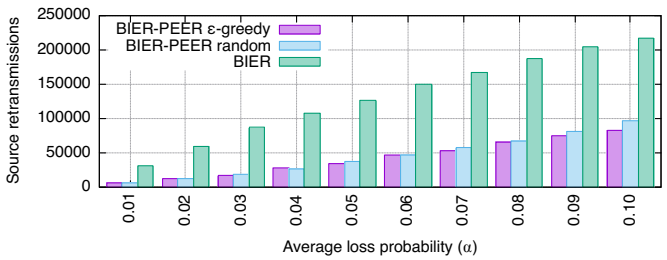
To simulate the existence of some local peers being "more suitable" than others, the link loss model is modified such that each ToR router has exactly one destination with a non-lossy ToR-to-destination link. An adaptive policy should then enable all other destinations connected to that ToR router to "learn" that this peer is the "most suitable" peer for retransmissions, and therefore, when detecting that a packet has not been received, send an SR-NACK towards first this "most suitable" peer, and only then to the source. To exemplify this, and to examine the intuitions from section 6.5.3, the ε-greedy policy has been implemented and tested against the *random peer selection* policy[6] (section 6.7.3) and standard *reliable BIER*. In order to allow for sufficient exploration, $\varepsilon = 0.2$ is used.

Simulations are run for different values of α, with results depicted in figure 6.13. As ε-greedy allows directing an SR-NACK towards peers that have a greater chance of being able to retransmit a packet, these are less often forwarded to the source, reducing the number of source retransmissions (figure 6.13b) and the link usage of the ingress link to the core router (figure 6.13a). Finally, when a SR-NACK is received by a "more suitable" peer, its retransmissions are more often successful

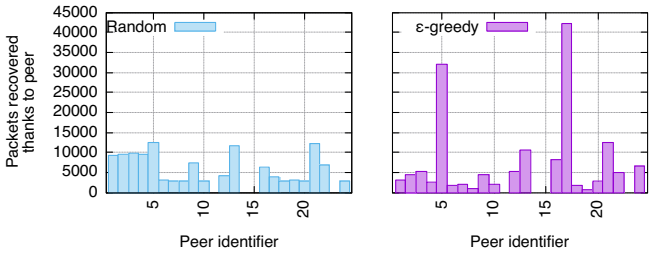
[6]The comparison is made to the random peer selection policy, rather than to any of the clustered peer selection policies. Indeed, selecting the destination with the non-lossy link for any of the clustering policies would amount to biasing in favour of that policy – selecting any other would amount to biasing in its disadvantage.



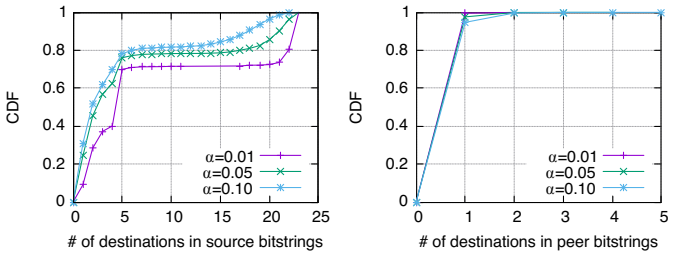
(a) Link usage between node 11 (New York) and its router



(b) Number of retransmissions by the source



(c) Histogram of peer contributions to recoveries ($\alpha = 0.10$), for random and ε-greedy strategies. Node 11 (NYC) is the source.



(d) Adaptive (ε-greedy) strategy in subtree: aggregation from source and peers.

Figure 6.14 – ISP topology: ε-greedy *vs.* random peer selection in subtree, for different values of the Gilbert-Elliott link loss probability α.

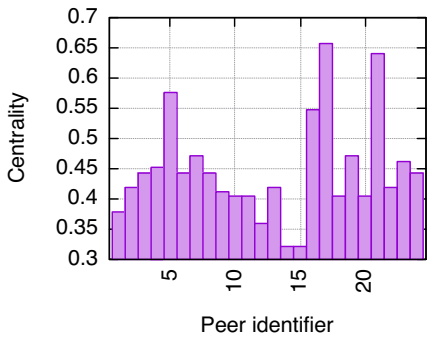


Figure 6.15 – Closeness centrality of the topology of figure 5.9.

(as it is connected to its ToR router over a non-lossy link), reducing subsequent retransmissions of that packet.

Figure 6.13c visualizes the behavior of the ε-greedy policy through plotting the number of times each peer is chosen by a destination as a candidate in an SR-NACK[7]. It can be observed that peers $0, 10, 20, 30$ (*i.e.,* the leftmost node of each rack, which is behind a non-lossy link) indeed contribute to most of the recoveries.

6.8 ISP Simulations

To validate the genericness of the proposed approach with respect to the network topology, another set of simulations has been conducted, this time in a real ISP topology. As compared to section 6.7, the resulting spanning tree is non-regular. Use cases for reliable multicast in such topologies include video streaming of live events.

[7]Only those times corresponding to the non-exploratory mode (where the destination actually selects its best peer) are retained. This allows subtracting the uniform noise due to the ε fraction of NACKs whose destinations are selected randomly.

6.8.1 Network Topology

As in section 5.4, the topology used for the purpose of this set of simulations is the "BT Europe, August 2010" topology extracted from the Internet Topology Zoo [150]. The topology is depicted in figure 5.9, and consists of 24 nodes. All nodes are located in Europe, except for nodes 11 (New York) and 12 (Washington DC), which are not visible on the figure, and have each a link to node 17 (London). As this topology features remote (transatlantic) nodes, it is suited to evaluate scenarios wherein it is costly to traverse certain links. As such, the scenario studied in this section will be the transmission of a multicast stream from node 11 (New York) to all other nodes.

Another point of interest of using such a non-regular topology is, that it provides a more natural framework to test adaptive peer selection policies. Indeed, some destination(s) will automatically be both close to the source (thus, more prone to have received packets because there are less links in the path to the source) and close to an important number of other peers. As an illustration, figure 6.15 shows the *closeness centrality* of each node in the graph, defined as the inverse of the mean distance to the other nodes. It can be observed that nodes 17 (London), 21 (Amsterdam) and 5 (Frankfurt) exhibit the highest centrality, meaning that they are 'local' to a large part of destinations, and would therefore be natural candidates to act as retransmitting peers.

6.8.2 Peer Selection Policies Evaluation

For the purpose of the simulations, a client is attached to each of the nodes, except for node 11 (NYC), to which a source is attached; the Dijkstra algorithm is first run offline to construct routing tables. As in section 6.7, links have 1 Gbps capacity, and the same simulation parameters are used. A multicast flow of 500 Mbps is sent from the source to each of the clients during four seconds, in 19 different simulations, for different values of the average link loss probability α. The *random selection policy* with one peerstring is used to evaluate the core properties of the mechanism introduced in this chapter. Additionally, the *ε-greedy policy* is evaluated, as a way to examine the convergence of destinations under adaptive policies towards highly-central nodes as retransmitters. Finally, as a baseline, simple reliable BIER as in chapter 5 is used.

Results are reported in figure 6.14. Figure 6.14a depicts the average usage of the link between the source and the router to which it is attached. Whereas with standard reliable BIER the link is saturated for high values of α ($\alpha \geq 6\%$), usage of this link is reduced with peer-based BIER retransmissions (the worst case being 720 Mbps for $\alpha = 10\%$), showing that the proposed mechanism allows protecting the source from having to retransmit too many packets. This can be also observed on figure 6.14b, depicting the number of individual packet retransmissions performed by the source: the number of source retransmissions falls by a factor of at least 2.2× when using peer-based retransmissions.

With respect to the comparison between static and adaptive policies, figure 6.14b shows that the number of source-based retransmissions is further reduced when using the adaptive policy. This can be explained by observing, for each peer, the number of packets successfully recovered through this peer, for the static policy and the adaptive policy (figure 6.14c) – the figures display the distribution for the lossiest tested scenario (corresponding to $\alpha = 0.1$), so as to better visualize the differences. Whereas with the static policy the distribution of retransmitters is relatively uniform, with the adaptive policy it can be clearly observed that peers 17 and 5 (and, to a lesser extend, peer 21) contributes to most of the recoveries. This confirms that retransmissions are handled by peers with high centrality (see figure 6.15), providing some aggregation of the retransmissions (as shown in figure 6.14d).

6.9 Summary of Results

This chapter extends the reliable multicast service based on BIER introduced in chapter 5. The proposed extension uses BIER for ensuring that data traffic is forwarded over minimal shortest path trees, and SR-based NACKs for reporting losses and requesting retransmissions. SR-NACKs allow failing destinations to contact an ordered set of peers for local recovery, before requesting retransmission from the source. All retransmissions are performed through BIER-enabled shortest paths.

The proposed protocol is compatible with standard BIER operation (RFC 8279 [8]): no caching is made by intermediate routers, retransmissions are regular BIER packets, and the retransmission

logic is handled exclusively by sources and destinations. In addition, a lightweight extension to the BIER forwarding plane is proposed (the processing of a peerstring in the BIER header), allowing destinations to automatically learn about potential candidates from which to ask retransmissions. In absence of this extension, peer-based recoveries can still be requested if destinations are manually configured to do so. By allowing for local repair of multicast failures, the proposed mechanism limits the amount of source retransmissions, and thus their impact in terms of network traffic and delay.

The proposed framework is generic enough to accommodate a broad spectrum of policies for selection of recovery peers, including static, adaptive, and operator-defined. Example of such policies are introduced and analyzed, both analytically and by way of network simulations. Evaluation suggests that substantial benefits in terms of increasing locality of recoveries and reducing usage of costly links can be achieved with relatively simple policies.

Part IV

Load-Balancing

Chapter 7

6LB: Scalable and Application-Aware Load Balancing with Segment Routing

Virtualization and containerization has enabled scaling of application performance by way of (i) running multiple instances of the same application within a data center, and (ii) employing a load-balancer for dispatching queries between these instances.

For the purpose of this chapter, it is useful to distinguish between two categories of load-balancers:

1. Network-level load-balancers, which operate below the application layer – a simple approach being to rely on Equal Cost Multi-Path (ECMP) [161] to homogeneously distribute network flows between application instances. This type of load-balancer typically does not take application state into account, which can lead to suboptimal resource utilization.

2. Application-level load-balancers, which are bound to a specific type of application or application-layer protocol, and make informed decisions on how to assign servers to incoming requests. This type of load-balancer typically incurs a cost from monitoring the state of each application instance, and sometimes also terminates network connections (such as is the case for an HTTP proxy).

A desirable load-balancer combines the best of these categories: *(i) be application or application-layer protocol agnostic* (*i.e.,* operate below the application layer) and *(ii) incur no monitoring overhead* – yet *(iii) make informed dispatching decisions depending on the state of the applications.*

Furthermore, data centers are more and more utilized to run virtualized network functions alongside traditional applications. In light of this, network load-balancers are more and more running as virtual functions (running for instance in virtual machines). This allows for the load-balancers themselves to take full advantage of the flexibility and redundancy of a virtualized data center: for resiliency, to allow a faulty load-balancer instance to be safely removed and replaced without incurring a service outage, or for scalability, by growing (and shrinking) the number of load-balancers to be able to accommodate different daily traffic demands and/or unexpected traffic peaks. The resulting architecture will thus distribute incoming flows between an edge router and several load-balancer instances, each of which will then redistribute the flows to application instances [162, 163]. A challenge arising from this architecture is to provide a consistent service when traffic for a given flow is directed by the edge router to a different load-balancer instance. This can happen *e.g.,* when a load-balancer instance is added or removed, causing the corresponding ECMP mapping between the edge router and the load-balancers to be updated correspondingly.

Thus in addition to the three desired properties exposed above can be added: *(iv) be able to be fully distributed*, providing the same service regardless of whether traffic is directed to different load-balancer instances within the lifetime of a flow.

These four objectives may appear irreconcilable: operating below the application layer makes it hard to take application state into account, and balancing by state rather than deterministically ties a flow to a given load-balancer instance. This chapter aims at providing a solution satisfying these four objectives, by challenging the traditional network paradigm wherein a packet is deterministically assigned only one destination.

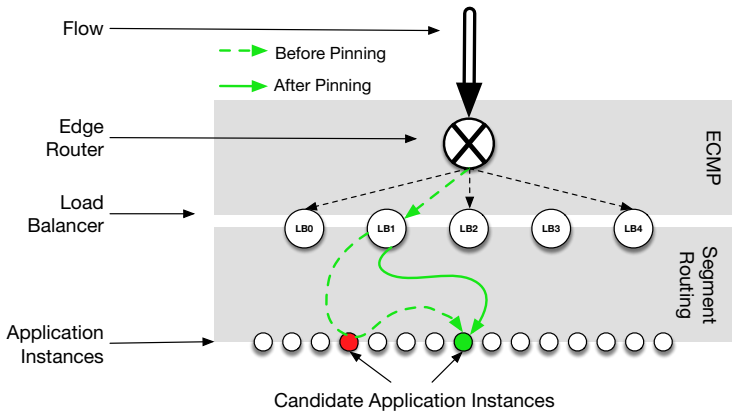


Figure 7.1 – 6LB architecture: load-balancers assign a flow to a set of candidate instances, through which the connection is passed until one accepts the connection (section 7.2). The flow is then pinned to the instance having accepted the connection (section 7.4).

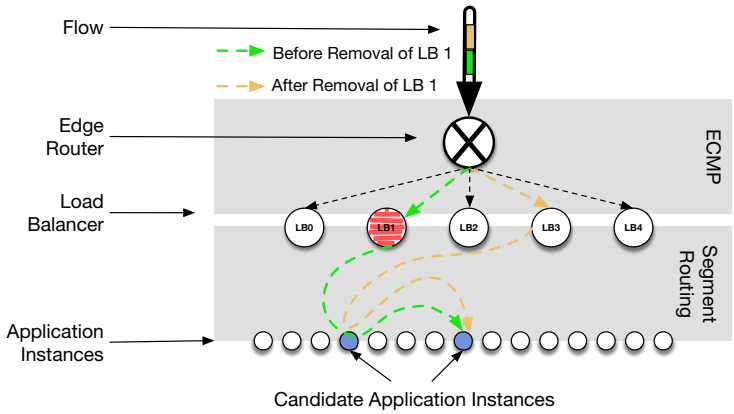


Figure 7.2 – 6LB consistent hashing: when a flow is rebalanced to another load-balancer instance by the edge router, consistent hashing (section 7.3) allows the flow to re-browse the same set of candidate instances, then to be re-pinned to the one that had accepted the connection (section 7.4).

7.1 Statement of Purpose

The purpose of this chapter is to propose 6LB, a load-balancing approach that is application-instance-load-aware, yet is both application and application-layer protocol independent and does not rely on centralized monitoring or transmission of application state.

A key argument behind this design goal is that an application instance itself is best positioned to know if it should be accepting an incoming query, or if doing so would degrade performance. Thus, 6LB disregards any design by which queries are unconditionally assigned to an application instance by the load-balancer. Rather, 6LB offers a received query to several candidate application instances, ensuring that exactly one instance accepts and processes the query.

The architecture behind 6LB is as follows, and as illustrated in figure 7.1: the edge router receives and uses ECMP to assign each incoming flow to a load-balancer. Each load-balancer selects a list of candidate application instances, to which it forwards the flow, using Segment Routing (introduced in section 1.2.1). In this way, 6LB enables that flow acceptance decisions are made strictly locally by an application instance, based on its real-time state information about itself, only – *i.e.,* without any centralized monitoring.

Once a flow is accepted by an application instance, it is "pinned" to it by the load-balancer: subsequent packets in that flow are all forwarded directly to that application instance. As depicted in figure 7.3, this is accomplished by the load-balancer inspecting the TCP handshake and establishing a mapping between a flow and the application instance serving it. A mechanism is provided

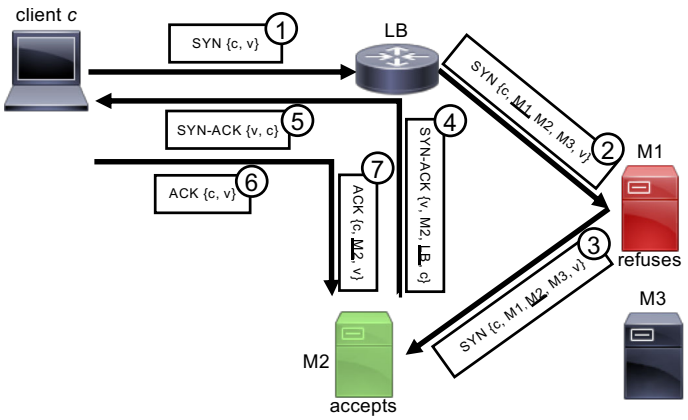


Figure 7.3 – TCP connection pinning from client c to application v, with 3 instances running on m_1, m_2, m_3. The path (source, segments, destination) is indicated between curly braces. The active segment is underlined.

to permit a load-balancer to recover this mapping, if for some reason it is lost[1].

The final mechanism in 6LB is consistent hashing, as illustrated in figure 7.2: a given flow will be assigned to the same set of candidate application instances, regardless of by which 6LB instance (past, current, or future, in case the pool of instances changes) it is assigned.

While the algorithms developed in this chapter are generally applicable for any number of candidate application instances, the concept of *power of two choices* [164] applies: selecting two candidate application instances ensures a low network footprint while providing a significant load-balancing improvement – with diminishing returns beyond that.

In addition to introducing the concept of directing queries through a chain of random candidate application instances with SR and letting them perform local connection acceptance decisions, the contributions of this chapter are: (i) a consistent hashing algorithm, (ii) an in-band stickiness protocol (the union of which allows to scale the number of load-balancer instances for reliability and performance), (iii) an analytic model analyzing the performance of SR-based load-balancing, (iv) experiments with the 6LB load-balancing architecture conducted on a large testbed, and finally (v) kernel-bypass implementations of 6LB and a state-of-the-art load-balancer (Maglev [162]) with performance comparison of the two implementations.

7.1.1 Related Work

Among existing load-balancing approaches below the application layer, Maglev [162] and Ananta [165] aim at providing a software load-balancer instance that can be scaled at will, and make use of ECMP to distribute flows between those instances. In addition to a flow stickiness table, they also make use of consistent hashing [166–168], for ensuring that data packets within a given flow are directed to the same application instance – regardless of the selected load-balancer instance forwarding a data packet, and with minimal disruption when the set of application instances changes. However, flows are distributed to application instances regardless of their current load. This is taken one step further in Duet [169] and Rubik [170], by moving the load-balancing function to hardware instances, while handing traffic over to software instances in case of failure. Conversely, [171, 172] use Software Defined Networking (SDN), with a controller monitoring the application instance load and network load, and then installing network rules to direct flows to these application instances. Spotlight [173] allows for weighted ECMP, with weights periodically updated by a centralized monitoring mechanism in order to reflect the current load of applications. Other frameworks, such as Trumpet [174], can be used to gather precise monitoring information on the network load.

Simple application-aware load-balancing policies (random, shortest queue, threshold) have been introduced in [175]; in [164, 176], it is shown that performing a load-balancing decision based on two random servers is sufficient to exponentially decrease the response time as compared to a random strategy. This concept has been used by [177] for peer-to-peer applications. Another

[1] For example, if a load-balancer is removed, and another load-balancer takes over the active flows it was serving.

similar idea, proposed in [178, 179], consists of duplicating queries among several replicas, and serving the quickest reply to the client. Somewhat similarly, in [64], SR is used to duplicate traffic through two different disjoint paths, so as to decrease latency and packet loss.

In [180], three load-balancing techniques are listed, which can be used for dispatching queries among Web servers: DNS round-robin, dispatchers that perform NAT or destination IP rewrite, and redirect-based approaches. Application-aware load-balancing mechanisms for static Web content include [181–183], which assign queries as a function of their estimated size so that each application instance becomes equally loaded. In [184], a feedback approach is used to estimate the parameters of a queuing model representing the system, before making a load-balancing decision.

Application-layer load-balancers, *e.g.,* HAProxy [185], also propose application-awareness by estimating the load on each application instance and assigning new queries accordingly. Load estimates are obtained either by tracking open connections through the load-balancer to the application instances (and thus do not consider other loads), or by periodically probing the backends for load information (and thus suffer from polling delay and incur network overhead). Another issue with application-level load-balancers is that network connections are reset when a failure causes a flow to be migrated from one load-balancer instance to another. The load-balancer introduced in [186] aims at solving this by keeping per-flow TCP state information in a distributed store.

7.1.2 Chapter Outline

The remainder of this chapter is organized as follows. Section 7.2 describes the use of Segment Routing for performing load-balancing. Section 7.3 describes a consistent hashing algorithm, which allows to distribute the load-balancing function into different instances, for scalability. Section 7.4 describes how "stickiness" can be established and recovered between load-balancer instances and application instances, using an in-band channel. An analytical performance model of 6LB is derived in section 7.5. 6LB is then experimentally evaluated in section 7.6, by way of a synthetic workload and a realistic workload consisting of a Wikipedia replica, and the performance of the implementation in terms of packet forwarding capabilities is evaluated. Finally, section 7.7 concludes this chapter.

7.2 Service Hunting with Segment Routing

This section describes how application-load-aware load-balancing can be performed, by way of a concept which will be referred to as *Service Hunting*.

7.2.1 Description

Service Hunting uses SR to direct network packets from a new flow through a set of *candidate application instances* until one accepts the connection. It assumes that applications are identified by *virtual IP addresses* (VIPs), and can be replicated among several instances, identified by their *physical addresses*. As introduced in section 1.4, machines run a *virtual router* (*e.g.,* VPP [31]), which dispatches packets between physical NICs and application-bound virtual interfaces. Finally, a load-balancer within the data center advertises routes for the VIPs.

When a new flow[2] for a VIP arrives at the load-balancer, the latter will select a set of candidate application instances from a pool, and insert an SR header identifying this set into the IPv6 data packet. The SR header will contain a list of segments, each indicating that the query can be processed by either of these application instances, and with the VIP as the last segment. Different policies can be used to select the list of candidate application instances to include in the SR header. It has been shown in [164] that selecting two random candidate application instances is enough to greatly improve load-balancing fairness, with a decreasing marginal benefit when using more than two instances. Thus, 6LB assigns each new flow to **two pseudo-randomly chosen** application instances – by way of a consistent hashing scheme, described in section 7.3.

When the flow reaches a candidate application instance, the corresponding segment in the SR header indicates that the virtual router may either forward the packet to the next segment, or may directly deliver it to the virtual interface corresponding to the application instance. This purely local decision of whether to accept a query is based on a policy shared *only* between the virtual router and the application instance, running on the same machine. To guarantee satisfiability,

[2]Typically, a TCP SYN packet as part of a connection request.

SC	Single-Choice policy (baseline)
SR_c	Static acceptance policy (Algorithm 7) with threshold c *e.g.,* **SR_4** is the policy of Algorithm 7 with $c = 4$
SR_{dyn}	Dynamic acceptance policy (Algorithm 8)

Table 7.1 – Notation

Algorithm 7 Static Connection Acceptance Policy **SR_c**

```
for each SYN packet do
    b ← number of busy threads
    if b < c or SegmentsLeft = 1 then
        SegmentsLeft ← 0, forward packet to application
    else
        SegmentsLeft ← SegmentsLeft − 1
        forward packet to next application instance in SR list
    end if
end for
```

however, the penultimate[3] segment indicates that the application instance must not refuse the query. It can be noted that this mechanism is not without similarities with the one introduced in chapter 3 for achieving zero-loss VM migration, in that the virtual router can decide to "skip" subsequent segments in the list (in chapter 3, a potential locator for a VM; here, a potential host for a workload), based on local decisions.

7.2.2 Connection Acceptance Policies

An application agent running within the virtual router decides whether the local application instance should to accept a flow. The application agent may make this decision based on whatever information it has locally available – from the operating system, or from real-time application metrics, if exposed. If information is exchanged with the server application software through shared memory, this incurs no system calls or synchronization, thereby imposing a negligible run-time cost.

This section describes two simple policies for deciding whether or not to accept new flows. They assume that the application uses a standard master-slave thread architecture. Section 7.6.1 will illustrate the application of these policies, in case of an HTTP server such as Apache. Table 7.1 summarizes the notation used throughout this chapter to designate the different policies.

Static (**SR_c**)

With n worker threads in the application instance, and a threshold parameter $c \in \{0, \ldots, n+1\}$, algorithm 7 describes a policy, **SR_c**, where an application instance accepts the flow if and only if strictly less than c worker threads are busy (except for the last in the SR list, which must always accept). Thus, an application instance which is "too busy" will be assigned a connection only if all previous application instances in the list are also "too busy". The choice of the parameter c directly influences the global system behavior: small values of c yield better results under light loads, and high values yield better results under heavy loads. As extreme examples, when $c = 0$, all requests are satisfied by the last application instance of their SR lists; when $c = n + 1$, all requests are satisfied by the first: both cases reduce to a random load-balancing scheme. If the chosen value of c is too small as compared to the load, almost all connections are treated by the last application instance of their SR lists, and vice-versa.

If the typical load is known, the value of c can be configured statically (by using the results of the analysis carried out in section 7.5) – otherwise, a dynamic policy can be employed.

Dynamic (**SR_{dyn}**)

When the typical load is unknown, the policy **SR_{dyn}** adapts c to maintain a rejection ratio of each application instance of $\frac{1}{2}$, as detailed in algorithm 8. Previous acceptance decisions are

[3]The ultimate segment is the VIP.

Algorithm 8 Dynamic Connection Acceptance Policy $\mathbf{SR_{dyn}}$

```
accepted ← 0, attempt ← 0
c ← 1    ▷ or other initial value
ε ← 0.1    ▷ or other increment value
windowSize ← 50    ▷ or other window size
for each SYN packet with SegmentsLeft = 2 do
    attempt ← attempt + 1
    if attempt = windowSize then
        ▷ end of window reached, adapt c if needed and reset window
        if accepted/windowSize < 1/2 − ε and c < n then
            c ← c + 1
        else if accepted/windowSize > 1/2 + ε and c > 0 then
            c ← c − 1
        end if
        attempt ← 0, accepted ← 0
    end if
    SR_c_policy()    ▷ use SR_c policy with current value of c
    if SR_c succeeded then
        accepted ← accepted + 1
    end if
end for
for each SYN packet with SegmentsLeft = 1 do
    SegmentsLeft ← 0, forward packet to application
end for
```

Protocol	new flow	pinned flow
IPv6 SR insert	72	56
IPv6 SR encap	96	80
IPv6 GRE Tunnel	88	44
IPv6 VXLAN Tunnel	140	70

Table 7.2 – Protocol overhead (in bytes) for different steering mechanisms, towards two (*new flow*) or one (*pinned flow*) application instances.

recorded over a fixed window of queries. When the end of the window is reached, if the number of accepted queries is significantly below (or above) $\frac{1}{2}$, the value of c is incremented (or decremented).

7.2.3 Protocol Overhead

Inserting an IPv6 SR header to direct a connection through multiple application instances has an impact in terms of packet size overhead. To quantify this, table 7.2 depicts the number of extra bytes needed to direct a packet through two (*new flow*) or one (*pinned flow*) application instances, for different protocols. As compared to other equivalent solutions allowing to direct a request through a set of instances (by sticking several successive tunneling headers), the proposed approach has the lowest overhead. After flow pinning, the overhead incurred by using SR as a steering mechanism is of 12 bytes as compared to GRE.

7.2.4 Reliability

The solution described in this section (and more generally in this chapter) focuses on the data-plane: it is assumed that a controller takes care of installing the mapping between a VIP and the set of addresses of machines hosting a corresponding application instance. Notably, as in other distributed load-balancing approaches [162,165], the controller should take care of health-checking the application instances, and removing them from the set of available instances when they are found unresponsive.

Since connection establishment packets go through a chain of instances rather than a single one, the properties of 6LB when facing failures need to be considered. Two scenarios can be distinguished. First, if a whole machine goes down (*critical failure*), new flows whose first candidate application instance is hosted on this machine will fail to be established, and new flows whose

second candidate application instance is hosted on this machine will fail only if the first instance rejected them. This incurs a p_R% failure overhead as compared to single-choice load-balancing approaches, where p_R is the percentage of connections being rejected by a first instance (*e.g.,* 50% with $\mathbf{SR_{dyn}}$). However, this happens only during the short amount of time before the controller detects that the machine is down and updates the backend pool on load-balancers accordingly[4]. Second, if an application instance goes down (crashes or becomes unresponsive) but the machine hosting it still remains up (*non-critical failure*), the virtual router on that machine will be able to forward connection establishment packets to the next instance in the SR list, for new flows whose first candidate instance is failing. Thus, for non-critical failures, 6LB *increases* the reliability of the system as compared to single-choice approaches, with a $(100 - p_R)$% failure reduction.

7.3 Horizontal Scaling with Consistent Hashing

Elastic scaling of the number of load-balancer instances is required, in order to accomodate dynamic data center loads and configurations [165]. When a load-balancer instance is added or removed, the ECMP function (see figure 7.1) performed by the edge router(s) may rebalance existing flows between remaining load-balancer instances. Thus, it is necessary to ensure that the mapping from flows to lists of candidate application instances is consistent across all load-balancers. This is achieved by the use of *consistent hashing*, depicted in figure 7.2 – which must also be resilient to modifications to the set of applications instances: adding or removing an application instance must have minimal impact on the mapping of existing flows.

Consistent hashing for load balancing is used for instance in Maglev [162], which proposes an algorithm mapping an incoming flow to one application instance. This section introduces a new consistent hashing algorithm (generalizing the one from Maglev) to allow each flow to map to an ordered list of application instances. Although 6LB uses 2 choices, the mechanism presented in this section is agnostic to this value, and is therefore presented for lists of C instances.

7.3.1 Generating Lookup Tables

With M *buckets* and N *application instances*, and where $N \ll M$, a pseudo-random permutation $p[i]$ of $\{0, \ldots, M-1\}$ is generated for each application instance $i \in \{0, \ldots, N-1\}$ – *e.g.,* by listing the multiples of the i-th generator[5] of the group $(\mathbb{Z}_M, +)$. These permutations are then used to generate a lookup table $t : \{0, \ldots, M-1\} \to \{0, \ldots, N-1\}^C$, mapping each bucket to a list of C application instances, following the procedure described in Algorithm 9. This table t is then used to assign SR lists of application instances to flows: each network flow will be assigned an SR list by hashing its network 5-tuple into a bucket j and taking the corresponding list $t[j]$.

Generating the lookup table t is done by browsing through the set of application instances in a circular fashion, making them successively "skip" buckets in their permutation until finding one that has not yet been assigned C application instances. Once each bucket has been assigned C application instances, the algorithm terminates. This process is illustrated in figure 7.4a, for $C = 2$ choices, with $N = 4$ application instances and $M = 7$ buckets. For each application instance i, the corresponding permutation table $p[i]$ is shown, where a circled number $\textcircled{j}_n$ means that bucket j has been assigned to that application instance at step n. For each bucket j, the lookup table $t[j]$ returned by the algorithm is also shown. For instance, bucket 3 is assigned to instance 1 (at step 5) and instance 2 (at step 6), thus the lookup table for bucket 3 is $(1, 2)$. The "skipping" behavior occurs *e.g.,* at step 9, where bucket 5 is skipped in $p[1][j]$ because it was already assigned two application instances.

7.3.2 Analysis

Resiliency

Figure 7.4b illustrates how this scheme is resilient to changes to the pool of application instances, by showing how removing application instance 0 modifies the tables $t[j]$ from the example of

[4] A simple way to improve the reliability of the system during this short amount of time would be to monitor, in-band, the responsiveness of the servers (*e.g.,* by gathering information about packet retransmissions or return traffic), and to rotate the order of SR lists for packets whose first instance is deemed unresponsive. This would increase reliability while ensuring that the browsed set of instances remains the one returned by consistent hashing.

[5] Defined as the $(i+1)$-st integer in $\{1, \ldots, M-1\}$ which is coprime to M.

Algorithm 9 Consistent Hashing

```
nextIndex ← [0, ..., 0]
C ← 2    ▷ or another size for SR lists
t ← [(−1, −1), ..., (−1, −1)]
n ← 0
while true do
    for i ∈ {0, ..., N − 1} do
        if nextIndex[i] = M then    ▷ permutation exhausted
            continue
        end if
        c ← p[i][nextIndex[i]]    ▷ advance in i's permutation
        ▷ skip buckets for which the SR list is already filled
        while t[c][C−1] ≥ 0 do
            nextIndex[i] ← nextIndex[i] + 1
            if nextIndex[i] = M then    ▷ permutation exhausted
                continue 2    ▷ continue the upper loop
            end if
            c ← p[i][nextIndex[i]]
        end while
        ▷ c is now the first bucket with SR list not filled
        choice ← 0
        while t[c][choice] ≥ 0 do
            choice ← choice + 1
        end while
        ▷ choice is now the first available position in the SR list
        t[c][choice] ← i
        nextIndex[i] ← nextIndex[i] + 1
        n ← n + 1
        if n = M × C then return t
        end if
    end for
end while
```

figure 7.4a. Assuming that flows are assigned to the first or second application instance in their SR lists with equal probability (as with the $\mathbf{SR_{dyn}}$ policy), the question is how flows mapped to a non-removed application instance ($1, 2, 3$ in this example) are affected by the table recomputation. For each bucket, one failure is counted for each non-removed application instance appearing in the lookup table before recomputation, but not after. In the example of figure 7.4, the only failure is induced by bucket 4, as the second entry of its lookup table, 1, does not appear in its newly computed lookup table, $(3, 2)$. With 10 non-removed flows, the failure rate in this example is thus 10%.

Intuitively, mapping flows to two application instances, instead of just to one, increases resiliency: it is less likely that the SR lists of a bucket before and after recomputation have empty intersection – for this to happen, a single bucket would need to be re-assigned twice.

The resiliency of algorithm 9 is studied by way of a simulation. An initial lookup table was computed. Then, k application instances were removed and the lookup table was recomputed – which allowed computing the previously introduced failure rate. The parameters were $N = 1000$ application instances, $M = 65537$ buckets, and 20 experiments were performed, for each value of k from 0 to 30.

Figure 7.5 reports the failure rate as a function of the number of removed instances. First, with $C = 1$ (*i.e.,* mapping each flow onto a single application instance), results identical to in [162, figure 12] are obtained, confirming that the algorithm reduces to the algorithm from [162] in this case. Using algorithm 9 for mapping each flow to two application instances ($C = 2$) shows up to 44% fewer failures (when $k = 8$) – or, to put it differently, 44% fewer TCP connections being reset.

Fairness

Each application instance picks the same number of buckets (as first or second entry), except potentially one in the last round. Assuming a probability of acceptance of $\frac{1}{2}$ (as with $\mathbf{SR_{dyn}}$), this guarantees that traffic is equally spread between application instances. Note that a given

Permutation tables $p[i]$ for each Application Instance i:

j	0	1	2	3	4	5	6
$p[0][j]$	④$_0$	⑤$_4$	⑥$_8$	0	1	②$_{12}$	3
$p[1][j]$	①$_1$	③$_5$	5	⓪$_9$	2	④$_{13}$	6
$p[2][j]$	⑤$_2$	③$_6$	①$_{10}$	6	4	2	0
$p[3][j]$	⑥$_3$	⓪$_7$	1	②$_{11}$	3	4	5

and lookup table t:

Bucket j	0	1	2	3	4	5	6
SR list $t[j]$	(3,1)	(1,2)	(3,0)	(1,2)	(0,1)	(2,0)	(3,0)

(a) Before removal of Application Instance 0

Permutation tables $p[i]$ for each Application Instance i:

j	0	1	2	3	4	5	6
$p[1][j]$	①$_0$	③$_3$	⑤$_6$	⓪$_9$	②$_{12}$	4	6
$p[2][j]$	⑤$_1$	③$_4$	①$_7$	⑥$_{10}$	④$_{13}$	2	0
$p[3][j]$	⑥$_2$	⓪$_5$	1	②$_8$	3	④$_{11}$	5

and lookup table t:

Bucket j	0	1	2	3	4	5	6
SR list $t[j]$	(3,1)	(1,2)	(3,1)	(1,2)	(3,2)	(2,1)	(3,2)

(b) After removal of Application Instance 0

Figure 7.4 – Example of permutation tables $p[i]$ and lookup table t ($C = 2, M = 7, N = 4$), before and after removal of application instance 0

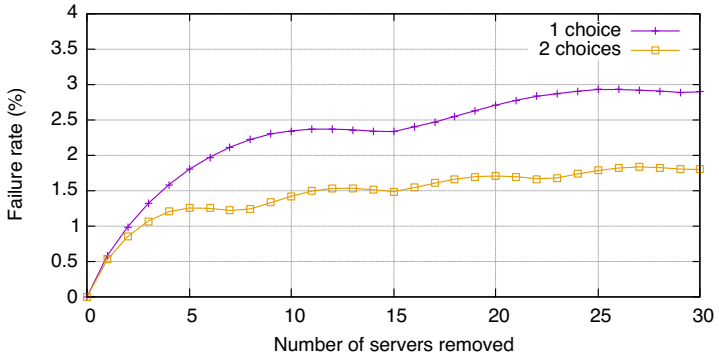


Figure 7.5 – Resiliency of consistent hashing to application instance removals: 1 choice (no SR) vs 2 choices (6LB)

application instance is not assigned the same number of first-choice buckets and second-choice buckets; this is nonetheless compensated by the fact that application instances do balance the load between themselves – this is evaluated in section 7.6.2.

Complexity

If permutations $p[i]$ are randomly distributed, this algorithm is a variant of the *coupon collector's problem*, and is expected to terminate in $M \log M + \mathcal{O}(M)$ steps for $C = 1$ [187], and in $M \log M + M \log \log M + \mathcal{O}(M)$ steps for $C = 2$ [188]. Hence, choosing two (rather than one) application instances requires only $1 + \frac{\log \log M}{\log M} \leq 1.368$ times more steps.

In comparison to the naïve algorithm consisting in building two uncorrelated lookup tables for the first and second application instances in the SR list, the benefit of using Algorithm 9 is twofold: the generation time is smaller, and jointly building the two entries make the scheme more resilient to changes as shown in figure 7.5.

State	Incoming SR function / SR functions added	Next state
`LB_LISTEN`	SYN from client / m_1`.connectAvail(`ℓ_1`)` m_2`.connectForce(`ℓ_1`)`	`HUNTING`
`LB_LISTEN`	data from client / m_1`.recoverStickiness(`ℓ_1`)` m_2`.recoverStickiness(`ℓ_1`)`	`HUNTING`
`HUNTING`	SYN from client / m_1`.connectAvail(`ℓ_1`)` m_2`.connectForce(`ℓ_1`)`	`HUNTING`
`HUNTING`	`createStickiness(`m`)` / remove SR header	`STEER(`m`)`
`STEER(`m`)`	data from client / `s.ackStickiness(`ℓ_1`)`	`STEER(`m`)`
`STEER(`m`)`	`removeStickiness(`m`)` / remove SR header	`LB_LISTEN` after 10 sec

Table 7.3 – Handshake protocol state machine for a given flow, at a load-balancer ℓ_1

7.4 In-band Stickiness Protocol

A load-balancer instance should, for each flow it handles, have knowledge of the application instance which has accepted the flow. First, this allows packets to be directly directed to the handling instance, without hopping through the chain of candidates (thereby reducing triangular traffic). Second, this ensures that, when the consistent hashing table is recomputed (*e.g.,* due to changes in the pool of applications), existing connections are protected against potential changes in the lookup table.

Thus, a signaling mechanism is required between the load-balancer and the application instances. Four properties should be satisfied: (i) no external control traffic should be generated, (ii) deep packet inspection should be minimized, (iii) incoming packets should go directly to the application instance handling the flow, and (iv) outgoing packets should not transit through the load-balancer. The latter property, called Direct Server Return (DSR) and introduced in [162,165], is crucial to the scaling of the load-balancer software: it enables it to treat only client-incoming packets, which are often more lightweight than server-emitted packets (acknowledgements *vs* data packets).

To satisfy (i) and (ii), SR headers are inserted into packets part of the accepted flow – *i.e.,* a set of SR *functions* are used for communicating between the load-balancer and the application instance. Objective (iii) is accomplished by having the appliation instance signal to the load-balancer when it has accepted a flow (by adding an SR header to, typically, the TCP SYN+ACK), and (iv) by making other traffic (packets other than, typically, the TCP SYN+ACK) bypass the load-balancer and be sent directly from the application instance to the client.

7.4.1 SR Functions

SR functions are used to encode actions to be taken by a segment endpoint, directly in the SR header. This is closely linked to how IPv6 addresses are assigned: since each machine is assigned a (typically, 64 bit [189]) IPv6 prefix, it is possible to use the lower-order bytes in this prefix to designate different functions, as recommended by the SR draft specification [50]. These functions will also depend on the address of the first segment in the SR list (the "sender" of the function). In practice, when a machine whose physical prefix is m receives a packet with SR header $(x, \ldots, m::f, \ldots)$, it will trigger a function f with argument x, which will be denoted by m`.f(`x`)`. In terms of a state machine, each SR function will thus (i) move the node from one state to another and (ii) trigger an action on the packet containing the SR function.

State	Incoming SR function / SR functions added	Next state
LISTEN	connectAvail(ℓ_1) (available) / remove SR header	WAIT(ℓ_1)
LISTEN	connectAvail(ℓ_1) (busy) / forward	LISTEN
LISTEN	recoverStickiness(ℓ_1) (not local) / forward	LISTEN
LISTEN	connectForce(ℓ_1) / remove SR header	WAIT(ℓ_1)
WAIT(ℓ_1)	connect[Avail|Force](ℓ_1) / remove SR header	WAIT(ℓ_1)
WAIT(ℓ_1)	data from app / ℓ_1.createStickiness(m)	WAIT(ℓ_1)
WAIT(ℓ_1)	ackStickiness(ℓ_1) / remove SR header	DIRECT(ℓ_1)
DIRECT(ℓ_1)	recoverStickiness(ℓ_2) (local) / remove SR header	WAIT($\ell_1$2)
DIRECT(ℓ_1)	ackStickiness(ℓ_1) / remove SR header	DIRECT(ℓ_1)
DIRECT(ℓ_1)	data from app / direct return to client	DIRECT(ℓ_1)
DIRECT(ℓ_1)	FIN from app / ℓ_1.removeStickiness(m)	LISTEN after 10 sec

Table 7.4 – Handshake protocol state machine for a given flow, at a machine m

7.4.2 Handshake Protocol

When a (TCP) flow is initiated, SR functions are added to the TCP handshake, so as to inform the load-balancer which candidate application instance has accepted the flow - thus establishing a handshake protocol between the load-balancer and the application instance handling a flow. This handshake protocol is formally described as state machines in tables 7.3 and 7.4, and detailed below:

1. Upon receipt of a flow (typically, a TCP SYN packet) from a client c for an application whose VIP is v, the load-balancer ℓ_1 will insert an SR header $(\ell_1, m_1 :: ca, m_2 :: cf, v)$ comprising the physical addresses m_1, m_2 of the machines hosting the two candidate application instances as given by the hashing function, and the original VIP. The suffix ca in the addresses indicates a function `connectAvail`, whereas cf represents a function `connectForce`. The first application instance in the list will make a local decision on whether to accept the flow. In case of refusal, the packet will be forwarded to the second application instance, which will have to forcefully accept the flow.

2. The (virtual router running in the) machine m_i ($i \in \{1, 2\}$) corresponding to the instance that has accepted the connection enters a *waiting* state for this flow. While in this state, it will temporarily direct traffic from the application towards the load-balancer, so that the latter can learn which application instance has accepted the connection. To do so, it inserts an SR header $(m_i, \ell :: cs, c)$ in packets coming from the application, where cs is a `createStickiness` function.

3. Upon receipt of such a packet, the load-balancer enters a *steering* state, during which traffic from the client to the application is sent using $(\ell, m_i :: as, v)$ as an SR header, as standing for a function `ackStickiness`. This permits both steering the traffic directly to the correct application instance, and acknowledging the creation of a stickiness entry.

4. Then, when (the virtual router running in) m_i receives such a packet, it enters a *direct return* state for this flow. As m_i has acknowledged the creation of the stickiness entry on the load-balancer, it thus does not need to send traffic through it anymore. Subsequent traffic of this flow will therefore be sent by m_i directly towards the client c, without using SR – thus allowing DSR.

5. Finally, when (the virtual router running in) m_i receives a connection termination packet from the application (typically, a TCP FIN or RST), it will insert an SR header $(m_i, \ell :: rms, c)$, where rms designates a `removeStickiness` function. This allows explicitly signaling connection

termination to the load-balancer. When receiving this packet, the load-balancer will start a small timer, at the expiration of which it will remove the corresponding stickiness entry – using a small timer ensures that packets are correctly directed to the rightful application instance while the transport layer connection teardown is happening. In addition to this explicit connection termination process, periodic garbage collection is used to remove stale entries from the load-balancer.

7.4.3 Failure Recovery

When adding or removing a load-balancer instance, traffic corresponding to a given flow might be redirected to a different load-balancer instance from the one over which it was initiated.

In order to recover state, when a new load-balancer instance receives a flow for which it does not have any state, incoming data packets corresponding to an unknown flow are added an SR header $(\ell, m_1::rs, m_2::rs, d)$, where rs is an SR function `recoverStickiness`. Consistent hashing ensures that $\{m_1, m_2\}$ is the same set as the one used by the previous load-balancer, with high probability[6]. When receiving a packet for this SR function, the application instance that, in the past, had accepted the flow, will re-enter the *steering* state, so as to notify the load-balancer. Conversely, an application instance that had previously not accepted the flow will simply forward the packet to the next application instance in the SR list.

7.5 Performance Analysis

In this section, an analytic model describing the performance of 6LB with the $\mathbf{SR_c}$ policy (algorithm 7) is derived. By way of this model, 6LB is compared to a **S**ingle-**C**hoice random flow assignment approach (**SC**), which reflects the behavior of standard consistent-hashing approaches (such as Maglev).

7.5.1 System Model

It is assumed that the system contains N application instances, with $N \to +\infty$, and that consistent hashing uses enough buckets such that the SR list associated with a flow is uniformly chosen amongst the N^2 possible lists.

In this section, the expected response time for the static acceptance policy $\mathbf{SR_c}$ described in algorithm 7 is derived, for c an integer threshold parameter. While a similar model has been considered in [190, section 4.4.4], the contribution in this chapter is that metrics of interest to 6LB (namely, the expected response time, the response time with forwarding delay, the fairness index, the probability of wrongful rejection, the response time distribution, and the reduction in number of servers) are derived, numerically computed, and validated against a real deployment experiment.

Incoming flows are assumed distributed according to a Poisson process of rate $\Lambda = N\lambda$, and each application instance offers an exponentially-distributed response time[7], with a processing rate μ **normalized to** $\boldsymbol{\mu = 1}$. For stability, the arrival rate must verify $\lambda < 1$. For $i \geq 0$, s_i is the fraction of application instances for which there are i or more pending flows (with $s_0 = 1$). This allows writing (as also found in [190]):

$$\begin{cases} \frac{\mathrm{d}s_i}{\mathrm{d}t} = \lambda(1+s_c)(s_{i-1}-s_i) - (s_i - s_{i+1}), \forall 1 \leq i \leq c \\ \frac{\mathrm{d}s_i}{\mathrm{d}t} = \lambda s_c(s_{i-1}-s_i) - (s_i - s_{i+1}), \forall i > c \end{cases} \tag{7.1}$$

When $i \leq c$, the probability of a flow being sent to an application instance which already handles $(i-1)$ other flows (i) directly is $(s_{i-1} - s_i)$ and (ii) after having being rejected by an application instance which already handles c or more flows is $s_c(s_{i-1} - s_i)$. This yields a total probability of $(1+s_c)(s_{i-1} - s_i)$. The same reasoning applies for $i > c$, where the probability of a flow being sent to an application instance which already handles $(i-1)$ flows $(i \geq 1)$ is the probability of having been rejected by a first application instance already handling c or more flows, before having been sent to an accepting application instance, yielding $s_c(s_{i-1} - s_i)$. The probability of a flow leaving

[6] This is not the case only when there is a *simultaneous* change in the pool of load-balancer instances *and* in the pool of application instances, and then only concerns those flows which correspond to consistent hashing failures and which, according to figure 7.5, amount to a few percents

[7] That is, the probability of a service lasting less than t is $1 - e^{-\mu t}$.

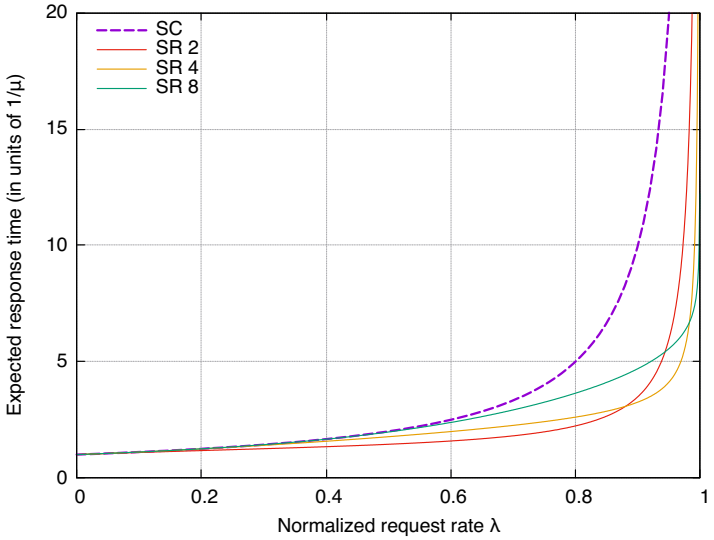


Figure 7.6 – Performance analysis of 6LB: mean response time $\mathbf{E}[T]$. **SC** vs $\mathbf{SR_2}$, $\mathbf{SR_4}$ and $\mathbf{SR_8}$.

an application instance already handling i flows is $(s_i - s_{i+1})$. Since the per-application-instance arrival rate is λ and the processing rate is $\mu = 1$, this yields equation (7.1).

To study the behavior of the system once in equilibrium, it is necessary to find a fixed point to the differential system (7.1). Setting $\frac{\mathrm{d}s_i}{\mathrm{d}t} = 0$ yields the following system of equations (where $s_0 = 1$):

$$\begin{cases} 0 = \lambda(1 + s_c)(s_{i-1} - s_i) - (s_i - s_{i+1}), \forall 1 \leq i \leq c \\ 0 = \lambda s_c(s_{i-1} - s_i) - (s_i - s_{i+1}), \forall i > c \end{cases} \tag{7.2}$$

For $1 \leq n \leq c$, summing equation (7.2) over $i = n$ to $+\infty$ gives $s_n = \lambda(1+s_c)(s_{n-1} - s_c) + \lambda s_c^2$, or: $s_n = \lambda[s_{n-1}(1+s_c) - s_c]$. Since $s_0 = 1$, this gives $s_1 = \lambda$. More generally, solving this recursion yields:

$$s_n = \frac{(1-\lambda)(\lambda(1+s_c))^n - \lambda s_c}{1 - \lambda(1+s_c)}, \forall 1 \leq n \leq c \tag{7.3}$$

Then, for $n > c$, summing equation (7.2) over $i = n$ to $+\infty$ gives $s_n = \lambda s_c s_{n-1}$, *i.e.*:

$$s_n = (\lambda s_c)^{n-c} s_c, \forall n \geq c \tag{7.4}$$

Plugging $n = c$ into equation (7.3) allows formulating an implicit equation for s_c:

$$s_c(1 - \lambda s_c) = (1 - \lambda)[\lambda(1 + s_c)]^c \tag{7.5}$$

This polynomial equation can be solved explicitly for $c \leq 5$, and numerically otherwise. Using this solution in equations (7.3) and (7.4) allows obtaining the value of s_n for any n, *i.e.,* the distribution of the number of flows awaiting being handled by an arbitrary application instance.

7.5.2 Expected Response Time

The expected number X of flows in the system can be computed, given that the probability that an application instance handles n flows is $(s_n - s_{n+1})$, as: $\mathbf{E}[X] = N \sum_{n=0}^{+\infty} n(s_n - s_{n+1}) = N \sum_{n=1}^{+\infty} s_n$.

According to Little's law [191], the expected time T spent in the system by a flow is: $\mathbf{E}[T] = \frac{\mathbf{E}[X]}{\Lambda} = \frac{1}{\lambda} \sum_{n=1}^{+\infty} s_n$. Summing s_n, as obtained in equations (7.3) and (7.4), and using equation (7.5) gives:

$$\sum_{n=1}^{c-1} s_n = \frac{\lambda - \lambda s_c(c-1) - s_c}{1 - \lambda(1+s_c)}, \sum_{n=c}^{+\infty} s_n = \frac{s_c}{1 - \lambda s_c}$$

Hence:

$$\mathbf{E}[T] = \frac{1 - cs_c(1 - \lambda s_c) - \lambda s_c(1 + s_c)}{(1 - \lambda s_c)(1 - \lambda(1 + s_c))} \tag{7.6}$$

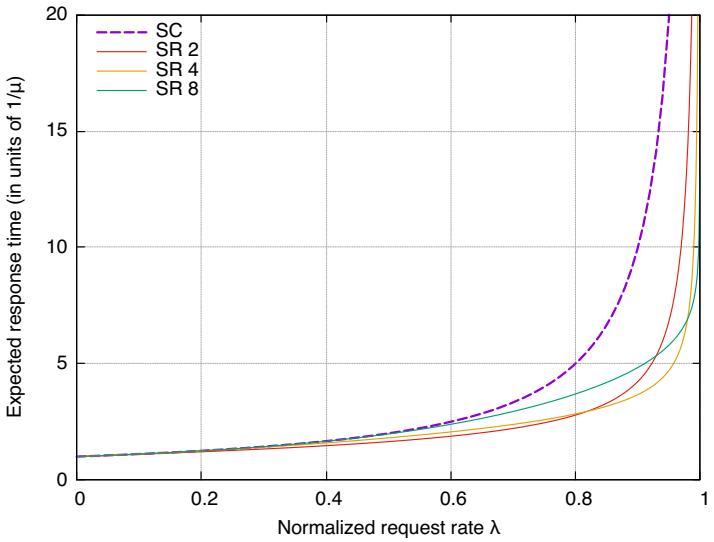


Figure 7.7 – Performance analysis of 6LB: worst-case response time with delay $\mathbf{E}[\hat{T}]$. **SC** vs $\mathbf{SR_2}$, $\mathbf{SR_4}$ and $\mathbf{SR_8}$.

Figure 7.6 depicts the value of $\mathbf{E}[T]$ for different $\mathbf{SR_c}$ policies, and for $\lambda \in [0, 1)$. As a reference, this value is compared to $\frac{1}{1-\lambda}$, the expected response time for **SC**, when clients are randomly assigned to *one* application instance. It can be observed that the $\mathbf{SR_c}$ policies uniformly yield an improvement over **SC**. When c is small, lower values of λ yield the highest gain; when c is important, higher values of λ yield the highest gain. For example, choosing $\mathbf{SR_8}$ offers an improvement over $\mathbf{SR_4}$ only when $\lambda \geq 0.983$, and thus might rarely be suitable.

7.5.3 Additional Forwarding Delay

In cases where the intra-data-center (server-to-server) forwarding delay is significant as compared to the job duration, forwarding a query to the second application instance in an SR list incurs an additional cost. If $\delta > 0$ denotes the network delay, multiplying this by the probability of being rejected by a first application instance gives the expected additional delay. Thus, the response time including delay, $\hat{T}$, verifies:

$$\mathbf{E}[\hat{T}] = \mathbf{E}[T] + \delta \times s_c \tag{7.7}$$

The following theorem states the conditions under which $\mathbf{SR_c}$ does not degrade performance as compared to **SC**:

Theorem 7.1. *As long as the network delay δ is smaller than the average job duration $1/\mu = 1$, the response time including delay with $\mathbf{SR_c}$ is better than with* **SC***:*

$$\mathbf{E}[\hat{T}] \leq \frac{1}{1-\lambda}, \quad \forall \delta \leq 1$$

Proof. Let $c \geq 1$ a threshold parameter, and $\lambda \in [0, 1)$. First, it will be shown that $s_n \leq \lambda^n$ for all $n \geq 0$. For $1 \leq n \leq c$, $s_n = \lambda[s_{n-1} - s_c(1 - s_{n-1})] \leq \lambda s_{n-1}$; for $n > c$, $s_n = \lambda s_c s_{n-1} \leq \lambda s_{n-1}$. Thus $s_n \leq \lambda s_{n-1}$ for all $n \geq 1$, and since $s_0 = 1$, it follows by induction that $s_n \leq \lambda^n$.

It remains to show that $\mathbf{E}[\hat{T}] \leq \frac{1}{1-\lambda}$. Let $\delta \in [0, 1]$, then: $\mathbf{E}[\hat{T}] = \frac{1}{\lambda}\sum_{n=1}^{+\infty} s_n + \delta s_c = \frac{1}{\lambda}\sum_{n=1}^{c-1} s_n + \frac{1}{\lambda}\frac{s_c}{1-\lambda s_c} + \delta s_c$. Using $s_n \leq \lambda^n, s_c \leq \lambda^c$ and $\delta \leq 1$ gives: $\mathbf{E}[\hat{T}] \leq \frac{1}{\lambda}\sum_{n=1}^{c-1} \lambda^n + \frac{1}{\lambda}\frac{\lambda^c}{1-\lambda^{c+1}} + 1 \cdot \lambda^c = \frac{1-\lambda^{c-1}}{1-\lambda} + \frac{\lambda^{c-1}}{1-\lambda^{c+1}} + \lambda^c = \frac{1}{1-\lambda} + \lambda^{c-1}(\frac{1}{1-\lambda^{c+1}} - \frac{1}{1-\lambda} + \lambda)$. Since $c \geq 1$, $\lambda^{c+1} \leq \lambda^2$, which yields: $\mathbf{E}[\hat{T}] \leq \frac{1}{1-\lambda} + \lambda^{c-1}(\frac{1}{1-\lambda^2} - \frac{1}{1-\lambda} + \lambda) = \frac{1}{1-\lambda} - \frac{\lambda^{c+2}}{1-\lambda^2} \leq \frac{1}{1-\lambda}$, which completes the proof. □

Figure 7.7 gives the expected response time including delay, in the "worst-case" in which the network delay equals the job duration ($\delta = 1$).

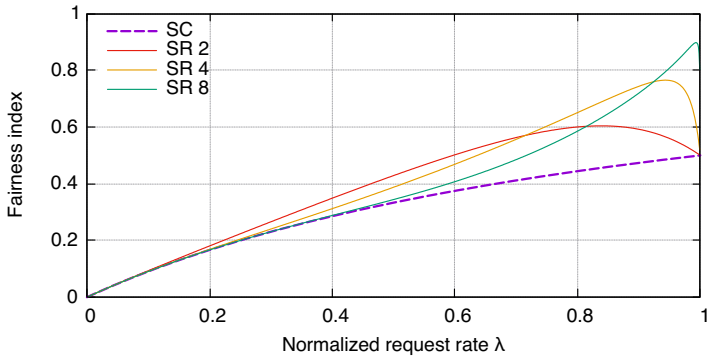


Figure 7.8 – Performance analysis of 6LB: fairness index $F = \frac{\mathbf{E}[X]^2}{\mathbf{E}[X^2]}$. **SC** vs $\mathbf{SR_2}$, $\mathbf{SR_4}$ and $\mathbf{SR_8}$.

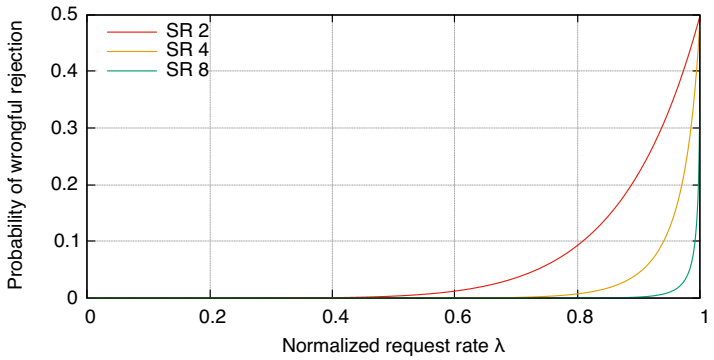


Figure 7.9 – Performance analysis of 6LB: probability of *wrongful rejection* for different $\mathbf{SR_c}$ policies.

7.5.4 Fairness Index

Jain's fairness index [192], defined as $F = \frac{\mathbf{E}[X]^2}{\mathbf{E}[X^2]} \in [0,1]$, is a measure for how even a load is distributed in a system: the closer it is to one, the more evenly is the load distributed.

Computing F requires computing $\mathbf{E}[X^2]$, the second moment of the number of flows in the system:

$$\mathbf{E}[X^2] = N\sum_{n=0}^{+\infty} n^2(s_n - s_{n+1}) = N\sum_{n=1}^{+\infty}(2n-1)s_n$$

Using equations (7.3), (7.4) and (7.5), this yields:

$$\sum_{n=1}^{c-1}(2n-1)s_n = \frac{1}{(1-\lambda(1+s_c))^2}\Big[\lambda^2(s_c+1)((c-1)^2 s_c+1) + \lambda s_c(2cs_c - 3s_c + 2c - c^2 - 2) - 2cs_c + \lambda + s_c\Big],$$

$$\sum_{n=c}^{+\infty}(2n-1)s_n = \frac{s_c(c(2-2\lambda s_c)+3\lambda s_c - 1)}{(1-\lambda s_c)^2}$$

Combining those expressions with the expression for $\mathbf{E}[X]$ from section 7.5.2 allows to compute F. Figure 7.8 depicts F, for different $\mathbf{SR_c}$ policies, and for $\lambda \in [0,1)$ – and compares with $\frac{\lambda}{1+\lambda}$, the fairness index for the **SC** policy. It can be observed that $\mathbf{SR_c}$ policies provide a better fairness than the reference **SC** policy, and that low values of c are more suitable for a low rate of new flow arrivals whereas high values of c are preferable for higher rates of new flow arrivals.

7.5.5 Wrongful Rejections

As has been shown in section 7.5.3, using the $\mathbf{SR_c}$ policy yields better performance than **SC**, because an overloaded application instance can offload a query to another random instance.

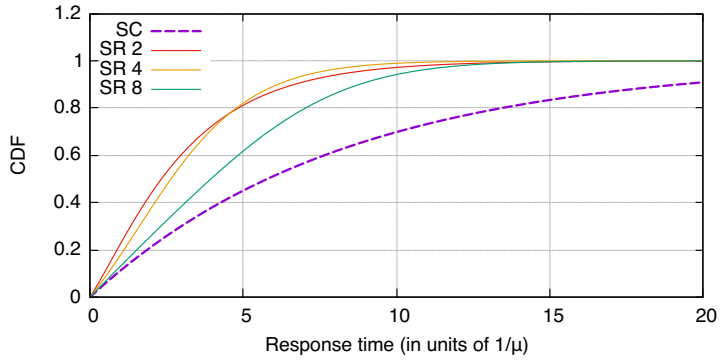


Figure 7.10 – Performance analysis of 6LB: CDF of response time for $\lambda = 0.88$. **SC** vs $\mathbf{SR_2}$, $\mathbf{SR_4}$ and $\mathbf{SR_8}$.

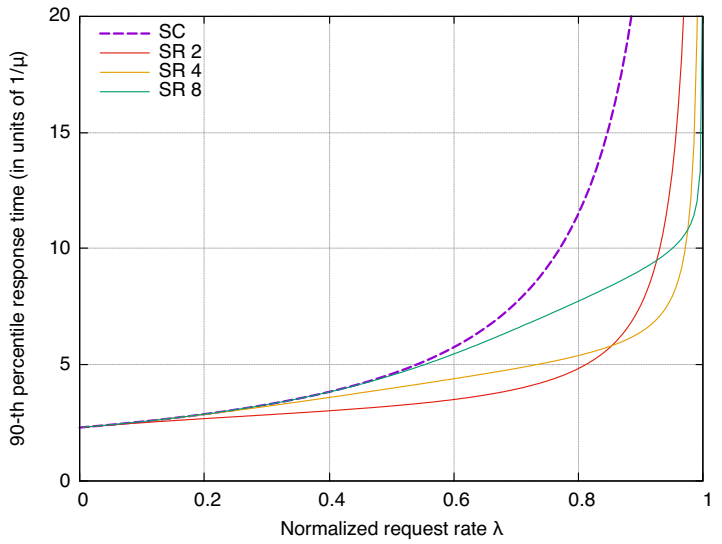


Figure 7.11 – Performance analysis of 6LB: 90-th percentile of response time. **SC** vs $\mathbf{SR_2}$, $\mathbf{SR_4}$ and $\mathbf{SR_8}$.

However, the proposed mechanism is not fully equivalent to the canonical "power-of-two-choices" scheme [164], wherein the least loaded of two random instances is chosen. Suboptimal decisions happen when a first instance handling $n \geq c$ flows rejects the connection to a second instance with strictly more than n flows. It is possible to estimate the quantity of such *wrongful rejections*: the probability of hitting such a pair of instances is $(s_n - s_{n+1})s_{n+1}$. Using equation (7.4), the probability p_w of wrongful rejection can be expressed as:

$$p_w = \sum_{n=c}^{+\infty}(s_n - s_{n+1})s_{n+1} = \frac{\lambda s_c^3}{1 + \lambda s_c} \tag{7.8}$$

In order to quantify this, figure 7.9 shows the probability of wrongful rejection for different $\mathbf{SR_c}$ policies. For example, with $\mathbf{SR_4}$, wrongful rejections happen with probability lower than 4.5% when $\lambda \leq 0.9$.

7.5.6 Response Time Distribution

The model also allows deriving the distribution of the time that a flow exists in the system. Knowing this distribution allows, for example, characterizing the performance of 6LB for Service Level Agreement (SLA) metrics, of the form *"No more than x% of clients should experience a response time $\geq y$"*.

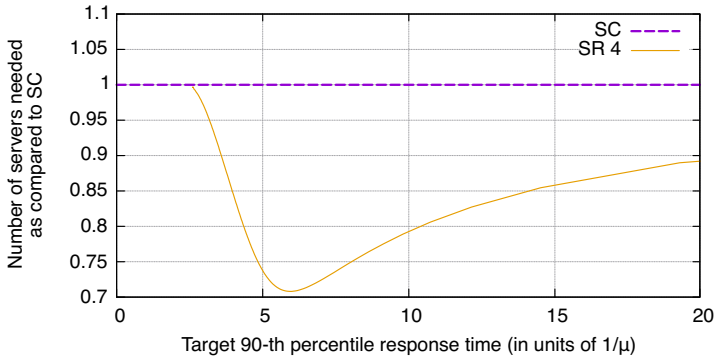


Figure 7.12 – Performance analysis of 6LB: reduction in number of instances when using $\mathbf{SR_4}$ vs **SC**, for different 90-th percentile SLAs

The distribution of the time T a flow waits, will be derived by computing its characteristic function $\varphi_T(\theta) = \mathbf{E}[e^{i\theta T}]$ (for $\theta \in \mathbb{R}$). Assume that the system is at its equilibrium given by equation (7.2), and that application instances use a FIFO policy with exponential response times. When a flow is being directed to an application instance that is already handling $(k-1)$ flows $(k \geq 1)$, its waiting time will be distributed as the sum of k independent and identically distributed (i.i.d.) exponential random variables $(\mathcal{E}_1, \ldots, \mathcal{E}_k)$ of parameter $\mu = 1$. The characteristic function of one such variable is $\mathbf{E}[e^{i\theta\mathcal{E}_1}] = \frac{1}{1-i\theta}$, hence $\mathbf{E}[e^{i\theta T}|k-1 \text{ clients}] = \left(\frac{1}{1-i\theta}\right)^k$.

When a flow arrives at the system, it will, with probability $(1_{k\leq c}+s_c)(s_{k-1}-s_k) = \frac{1}{\lambda}(s_k - s_{k+1})$, be directed to an application instance which is already handling $(k-1)$ other flows. Based on this, it is possible to express the characteristic function of the waiting time of an arbitrary flow:

$$\mathbf{E}[e^{i\theta T}] = \sum_{k=1}^{+\infty} \frac{1}{\lambda}(s_k - s_{k+1}) \left(\frac{1}{1-i\theta}\right)^k$$

Using equations (7.3), (7.4) and (7.5), this can be expressed as:

$$\mathbf{E}[e^{i\theta T}] = \frac{(1-\lambda)(1+s_c)}{1-\lambda(1+s_c)-i\theta} - \frac{s_c(1-\lambda s_c)}{(1-\lambda s_c - i\theta)(1-\lambda(1+s_c)-i\theta)(1-i\theta)^{c-1}} \tag{7.9}$$

which can be inverted to find p_T, the probability density of T, using $p_T(t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{-i\theta t}\mathbf{E}[e^{i\theta T}]\mathrm{d}\theta$.

Figure 7.10 depicts the CDF of this probability distribution, for $\lambda = 0.88$, and for various $\mathbf{SR_c}$ policies. At this high load, the distribution for the $\mathbf{SR_c}$ policies exhibit lower response times and less variance as compared to **SC**.

Integrating this probability density allows finding π_x, the x-th percentile of response time, defined as the number satisfying:

$$\mathbf{P}[T \leq \pi_x] = \int_0^{\pi_x} p_T(t)\mathrm{d}t = \frac{x}{100} \tag{7.10}$$

Figure 7.11 depicts π_{90}, the 90-th percentile of response time, for various $\mathbf{SR_c}$ policies and for $\lambda \in [0, 1)$, and is compared to $\frac{\ln(10)}{1-\lambda}$, the same metric for the **SC** policy. Similarly as in figure 7.6, the response time with $\mathbf{SR_c}$ is lower than with **SC**, and small values of c are more suitable for low request rates.

7.5.7 Reducing the Number of Application Instances

The developed model allows estimating the gain, in terms of how many fewer application instances are required to attain a certain SLA, when using 6LB as compared to when using "plain" **SC**. Assume that a system faces a daily request rate profile with a peek rate Λ_0, and that the goal is to provide a given SLA μ_0 on the 90-th percentile of response time: no more than 10% of clients should receive a target response time greater than μ_0 (*i.e.*, $\pi_{90} = \mu_0$).

With a simple **SC** load-balancer, the system faces a normalized request rate of $\lambda_0 = \Lambda_0/N$, and the 90-th percentile of response time is $\pi_{90} = \frac{\ln(10)}{1-\Lambda_0/N}$ – thus, requiring deploying $N = \frac{\Lambda_0}{1-\ln(10)/\mu_0}$ application instances to meet the SLA.

As per equation (7.10), let $\pi(\lambda)$ be the function giving the 90-th percentile of response time π_{90} as a function of λ, when using 6LB with the $\mathbf{SR_c}$ policy. In order to meet the SLA, *i.e.,* to ensure that $\pi(\Lambda_0/N) = \mu_0$, $N' = \frac{\Lambda_0}{\pi^{-1}(\mu_0)}$ application instances must be deployed.

Comparing **SC** and 6LB with $\mathbf{SR_c}$ yields:

$$\frac{N'}{N} = \frac{1 - \ln(10)/\mu_0}{\pi^{-1}(\mu_0)} \tag{7.11}$$

Figure 7.12 depicts this reduction in number of application instances, as a function of the target SLA μ_0, between **SC** and $\mathbf{SR_4}$. If the SLA requires that no more than 10% of clients experience a response time greater than *e.g.,* $\mu_0 = 6$, then if that is met by a deployment of *e.g.,* $N = 100$ application instances when using **SC**, only $N' = 71$ application instances will be required if using 6LB with the $\mathbf{SR_4}$ policy.

7.6 Evaluation

This section describes an evaluation of 6LB on real software. Section 7.6.1 introduces the implementation and experimental perform; then, section 7.6.2 reports results of tests conducted on a synthetic workload, and section 7.6.3 on a realistic workload.

7.6.1 Experimental Platform

The experimental platform used for evaluating 6LB is composed of a load-balancer and a server agent for the Apache HTTP server.

Load-Balancer

The load-balancer performing consistent hashing, SR header insertion and flow steering is implemented as a VPP plugin [31]. Having kernel-bypass capabilities and embedding an IPv6 Segment Routing stack, VPP is a suitable choice to build a performing implementation. As a reference, Maglev [162] was also implemented to evaluate the single-choice consistent hashing flow assignment policy **SC**.

Apache HTTP Server Agent

A server agent for the Apache HTTP server [193] has been implemented as a VPP plugin, accessing Apache's *scoreboard* shared memory[8] to allow the virtual router to access the state of the application instance. Apache uses a *worker thread* model: a pool of worker threads is started in advance, and received queries are dispatched to those threads. Thus, a simple exposed metric is the state of each worker thread, allowing to count the number of busy/idle threads, and use this to decide on connection acceptance, using one of the policies described in section 7.2.2.

System platform

The experiments described in sections 7.6.2 and 7.6.3 are conducted on a common platform. An edge router and two load-balancer instances are deployed as 2-core VMs residing in one physical machine. $N = 48$ application instances of an Apache HTTP server reside each in a 2-core VM, all of which are hosted across 4 physical machines (distinct from the one hosting the edge router/load-balancers). The edge router is configured to split traffic for the application across the two load-balancer instances, by way of ECMP, as in figure 7.1. VPP instances running in the edge router VM, in the load-balancer VMs, and in each of the VMs of the application instances, are on the same Layer-2 link, with routing tables statically configured. Each physical machine has a 24-core Intel Xeon E5-2690 CPU.

[8] This shared memory, internal by default, can be exposed as a named file by specifying the `ScoreBoardFile` directive in the server configuration.

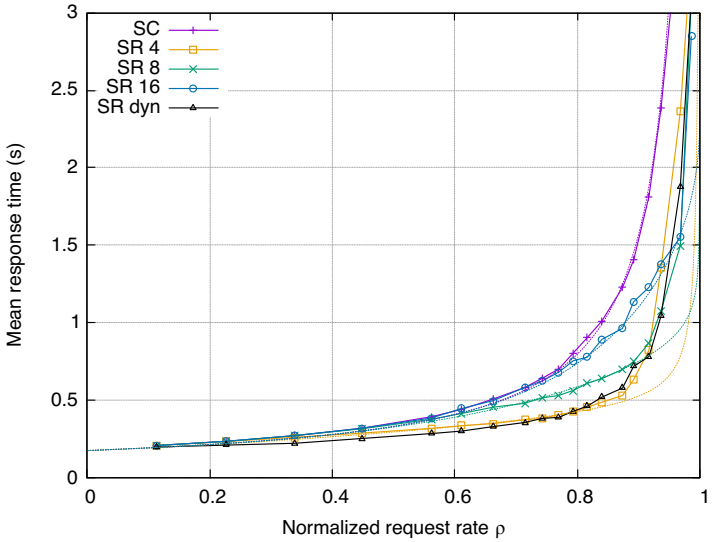


Figure 7.13 – Connection acceptance policies evaluation: average page load time. **SC** vs $\mathbf{SR_4}$, $\mathbf{SR_8}$, $\mathbf{SR_{16}}$, $\mathbf{SR_{dyn}}$.

The size of the consistent hashing table of the load-balancer instances was set to $M = 65536$ (except for the experiments of figures 7.17 and 7.18), and the Apache servers were configured to use the `mpm_prefork` module, each with 32 worker threads and with a TCP backlog of 128.

The `tcp_abort_on_overflow` parameter of the Linux kernel was enabled, triggering a TCP RST when the backlog of TCP connections exceeds queue capacity, rather than silently dropping the packet and waiting for a SYN retransmit. Thus under heavy load, it is application response delays that are measured, and not possible TCP SYN retransmit delays.

7.6.2 Poisson Traffic

Traffic and Workload Patterns

To evaluate the efficiency of the connection acceptance policies from section 7.2.2 under different loads, 6LB was tested against a simple CPU-intensive web application, consisting of a PHP script running a `for` loop with an exponentially distributed number of iterations, and whose duration is 190 ms in average. Using such a distribution ensures that job durations exhibit reasonable variance (the standard deviation of the exponential distribution is equal to its mean). A traffic generator sends a Poisson stream of queries (HTTP requests), with rate λ. A bootstrap step consisted of identifying λ_0, the maximum rate sustainable by the 48-instances farm, *i.e.,* the smallest value of λ for which some TCP connections were dropped.

Connection Acceptance Policies Evaluation

With $\rho = \lambda/\lambda_0$ as the normalized request rate, for 20 values of ρ in the range $(0, 1)$, a Poisson stream of 80000 queries with rate ρ was injected in the load-balancers, using the policies $\mathbf{SR_4}$, $\mathbf{SR_8}$, $\mathbf{SR_{16}}$, and $\mathbf{SR_{dyn}}$. As baseline, the same tests were run with a policy **SC** where queries are pseudo-randomly assigned to one application instance, without Service Hunting, using the single-choice consistent hashing algorithm of Maglev [162].

Figure 7.13 depicts mean response times for each tested request rate and for each policy, and show that, among those, $\mathbf{SR_4}$ yields the best response time profile, up to 2.3× better than **SC** for $\rho = 0.87$. $\mathbf{SR_8}$ and $\mathbf{SR_{16}}$ likewise perform better than **SC** for all loads, but with a lesser impact. $\mathbf{SR_{dyn}}$ offers results close to the best tested static policy. In order to validate the analytical model introduced in section 7.5, the response time as obtained from equation (7.6) is displayed in dotted lines alongside the experimental results[9]: it can be observed that the model accurately fits the data, as long as $\rho < 0.9$. After that, the assumptions (steady state, infinite number of servers) do not hold anymore.

[9] A fit is performed on **SC** to rescale the units. The obtained scaling coefficients are then used for all policies.

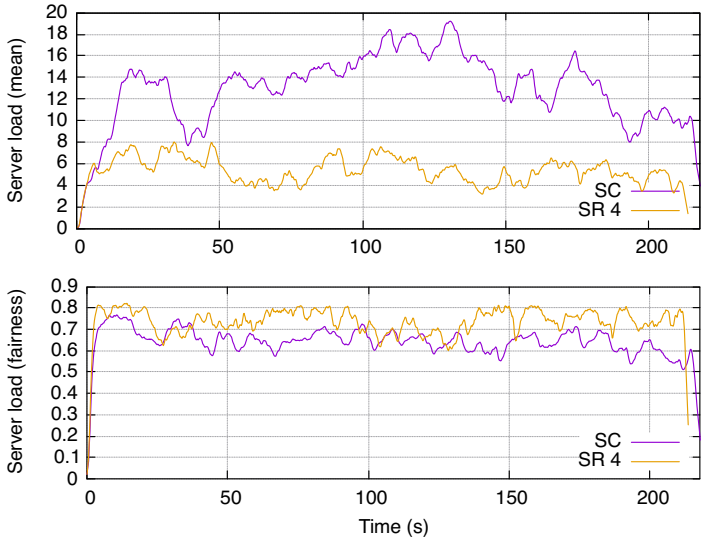


Figure 7.14 – Connection acceptance policies evaluation: instantaneous server load, $\rho = 0.89$. **SC** vs $\mathbf{SR_4}$.

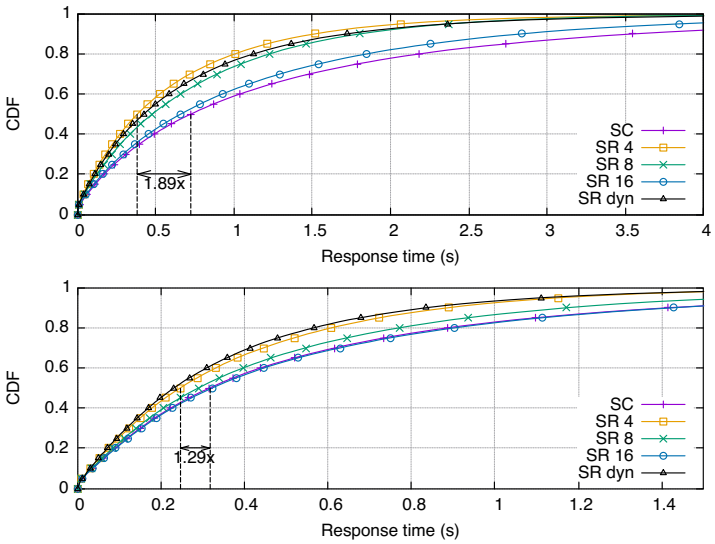


Figure 7.15 – Connection acceptance policies evaluation: CDF of page load time: $\rho = 0.71$ (top) and $\rho = 0.89$ (bottom). **SC** vs $\mathbf{SR_4}$, $\mathbf{SR_8}$, $\mathbf{SR_{16}}$, $\mathbf{SR_{dyn}}$.

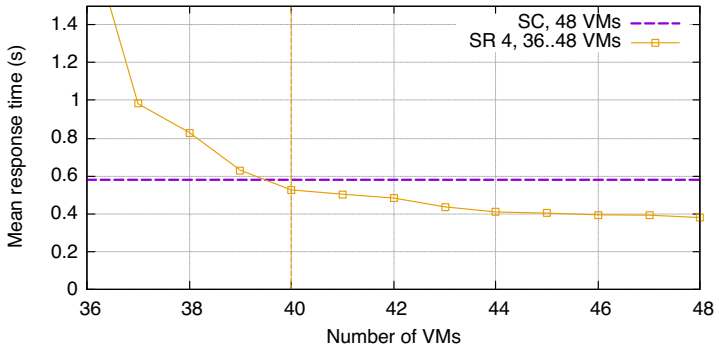


Figure 7.16 – Average page load time while decreasing the number of VMs: $\mathbf{SR_4}$, $\rho = 0.71$.

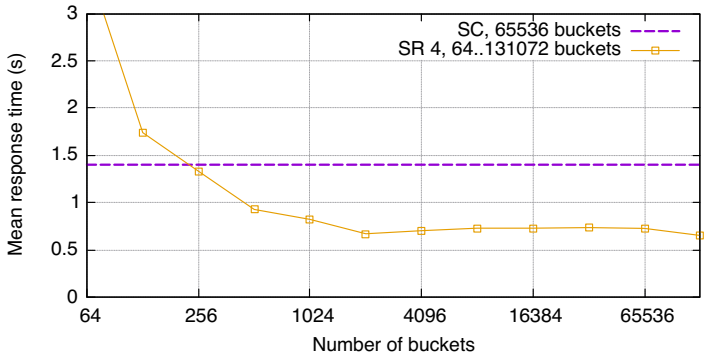


Figure 7.17 – Influence of the consistent hashing table size M: $\mathbf{SR_4}$, $\rho = 0.89$.

Figure 7.15 shows the CDF of the page response time for the 80000 queries batch with $\rho = 0.89$, for each policy. **SC** exhibits a very dispersed distribution of response times, whereas the different $\mathbf{SR_c}$ policies yield lower, and less dispersed, response times. This can be explained by inspecting the evolution of the mean instantaneous load (the number of busy worker threads) over all application instances, as well as the corresponding fairness index: $\frac{(\sum_{i=1}^{48} x_i(t))^2}{48 \sum_{i=1}^{48} x_i(t)^2}$ (where $x_i(t)$ is the load of server i at time t), depicted in figure 7.14[10]. As $\mathbf{SR_4}$ better spreads queries between all servers (the fairness index is closer to 1), and servers are individually less loaded, better response times result.

For lighter loads, a similar behavior can be observed, except that $\mathbf{SR_c}$ policies for high values of c exhibit no benefits as compared to **SC**. Figure 7.15 shows the CDF of the page load time for an experiment where $\rho = 0.71$: $\mathbf{SR_{16}}$ yields no improvement over **SC**, and $\mathbf{SR_8}$ yields a relatively small improvement, however the $\mathbf{SR_4}$ policy provides a substantial improvement in response times – and $\mathbf{SR_{dyn}}$ remains able to successfully match $\mathbf{SR_4}$, the best tested static policy.

Reducing the Number of Application Instances

Previous experiments have shown how 6LB is able to yield a reduced page response time, for a given request rate. Conversely, if an SLA on the target response time is to be satisfied, 6LB can be used to decrease the number of application instances needed to reach that SLA. In order to quantify this, a simple experiment has been conducted to find out how many VMs can be shut off while achieving a pre-defined SLA. Assume that the 48 VMs were deployed with **SC** to attain an average response time of 0.58 s, *i.e.,* that the application faces a total request rate of $\rho = 0.71$ (values taken from figure 7.13). Using the same request rate, a batch of 80000 requests was ran against less and less VMs with the $\mathbf{SR_4}$ policy, until the same average response time was reached. Figure 7.16 shows the average response time as a function of the number of VMs: with 6LB, 40 VMs are needed to meet the same SLA as compared to 48 VMs with **SC** – a reduction of 17%.

Influence of the Consistent Hashing Table Size

Using a smaller hash table can be beneficial in environments with tight resources, but at the cost of evenness in the distribution of the application instances within first segments of the SR lists (as explained in section 7.3). In order to quantify this, a Poisson stream of 80000 requests with request rate $\rho = 0.71$ was sent to the load-balancers against the $\mathbf{SR_4}$ policy, using different hash table sizes. Figure 7.17 shows the average response time as a function of the table size used (tables have sizes 2^k for performance reasons). The response times are almost identical for high table sizes, with a noticeable influence when $M \leq 1024$. Also, except when $M \leq 128$, the average response time stays lower than when using the **SC** policy with the same rate.

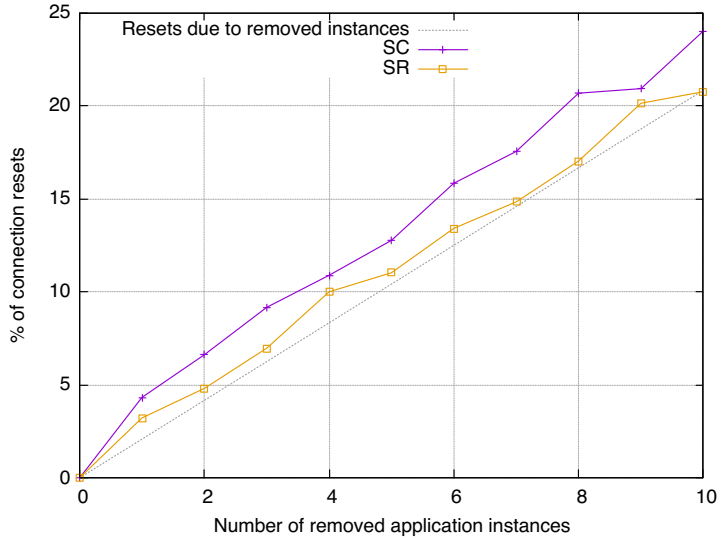


Figure 7.18 – Connection resets when removing x application instances and simultaneously switching to another 6LB instance

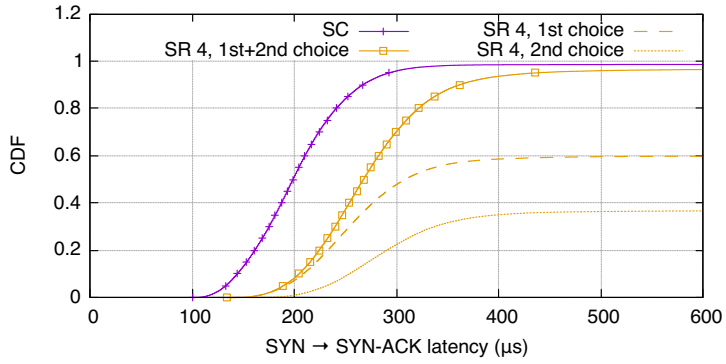


Figure 7.19 – SYN → SYN-ACK latency. $\mathbf{SR_4}$, $\rho = 0.71$.

Consistent Hashing Resiliency

The resiliency of the consistent hashing mechanism introduced in Algorithm 9 in real conditions is evaluated through a simple experiment, where a simultaneous change in the application instances pool and in the load-balancer pool is introduced. With $M = 4096$ buckets in the consistent hashing tables, 1000 long-lived flows are injected in the system and handled by the first 6LB instance. Then, x application instances are removed, while at the same time the ECMP router is reconfigured to use the second 6LB instance. The number of connection resets is recorded for $\mathbf{SR_8}$ and **SC**, and depicted in figure 7.18 for several values of x (averaged over 10 experiments). 6LB increases the resiliency over **SC** (as described in section 7.3.2): apart from "unavoidable" resets corresponding to connections that were pinned to a removed instance, no more than 2% of extra connections were reset by 6LB, as compared to 4% with **SC**.

SYN → SYN-ACK Latency

In order to quantify the additional forwarding latency induced by 6LB, figure 7.19 depicts the SYN → SYN-ACK latency as seen by the client for $\mathbf{SR_4}$ and **SC**, for the experiment where $\rho = 0.71$. As compared to **SC**, with 6LB, the SYN packet *can* be forwarded to an extra candidate instance, and the SYN-ACK packet *must* be forwarded through the load-balancer. Overall, this increases the median latency by 69 μs. Restricted to those connections that are accepted by the

[10] These values have been smoothed through an Exponential Window Moving Average filter, of parameter $\alpha = 1 - \exp(-\delta t)$ where δt is the interval of time in seconds between two successive data points.

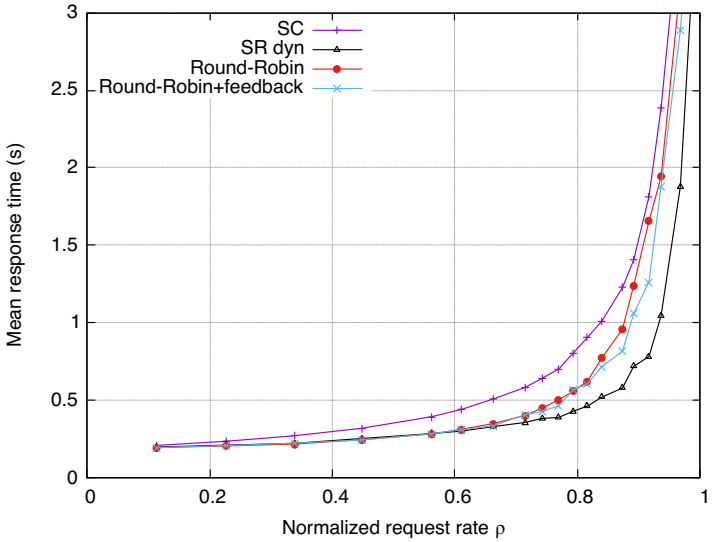


Figure 7.20 – Average page load time for centralized policies

second candidate in the SR list (corresponding to 38% of the 80000 queries in this experiment), the median latency is increased by 57 μs. For connections accepted by the first instance, the median latency is increased by 32 μs.

Comparison against Centralized Policies

Centralized load-balancing policies do not offer the resiliency of consistent hashing approaches, but in exchange provide more fairness. In order to position 6LB as compared to this class of load-balancers, two centralized policies are evaluated: (i) Round-Robin and (ii) weighted Round-Robin with feedback [173]. With the latter policy, feedback is obtained by polling the load of each application instance every 200 ms (over an out-of-band TCP channel), before adjusting the weight of the instance in the Round-Robin algorithm accordingly[11]. Figure 7.20 depicts the average page load time as a function of the request rate ρ. Results show that Round-Robin provides more fairness than single-choice consistent hash, but is outperformed by $\mathbf{SR_{dyn}}$ (with equivalent results for light loads $\rho \leq 0.7$). For heavier loads, the feedback policy slightly improves performance over Round-Robin, but remains outperformed by $\mathbf{SR_{dyn}}$: this shows the benefit of using instantaneous information rather than relying on periodic feedback.

Influence of the Variance of Service Times

To understand the influence of the variability of job service times, an experiment is conducted, where job CPU times distributions have different variances. To that purpose, the previously used exponential distribution is replaced with several log-normal distributions[12]. The response times are set to have the same median as previously, but different variance parameters, allowing to evaluate from constant to very skewed response times. Figure 7.21 depicts the mean response time as a function of the coefficient of variation[13] of service times, for a Poisson stream of 80000 queries at rate $\rho = 0.71$, against $\mathbf{SR_{dyn}}$, **SC**, and Round-Robin.

In the extreme case where response times are constant, Round-Robin performs the best (as instances will have totally processed a query before being assigned a new one) and 6LB performs better than **SC**. Indeed, with single-choice consistent-hashing queries can be placed on a server that is already busy (if "unlucky once"), whereas 6LB needs to be "unlucky twice" for this to happen.

[11] The weight w is adjusted with $w = 0.1 + 0.9\exp(-8 \times (b/32)^2)$, where b is the current number of busy worker threads.

[12] Log-normal distributions are a simple class of positive-valued distributions with a parameter influencing the variance. A log-normal distribution with parameters μ, σ has density $\frac{1}{x\sigma\sqrt{2\pi}} \exp[-(\log x - \mu)^2/(2\sigma^2)]$, median e^{μ} and variance $(\exp(\sigma^2) - 1)\exp(2\mu + \sigma^2)$.

[13] The *coefficient of variation* of X is the standard deviation of X divided by its average.

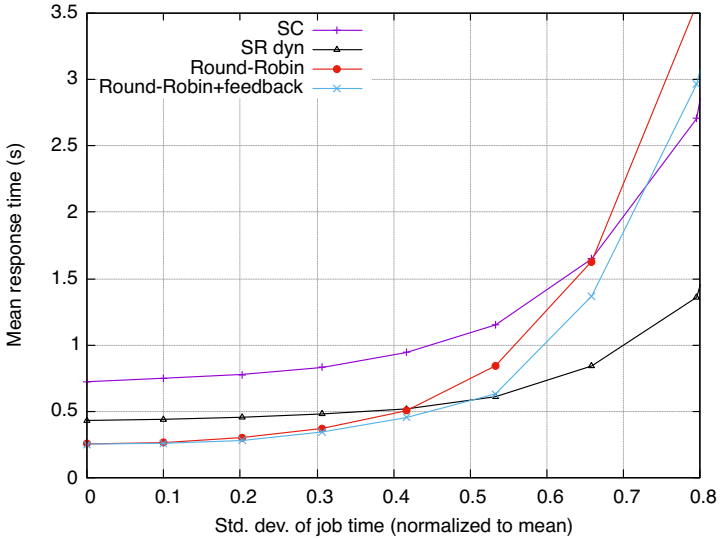


Figure 7.21 – Average page load time for different variances of job times, $\rho = 0.71$.

When the skewness of job service times increases, 6LB's use of local information becomes a greater and greater advantage, and it eventually shows the best performance among all approaches.

7.6.3 Wikipedia Replay

To evaluate the efficiency of 6LB when exposed to a realistic workload, an experiment has been constructed to reproduce a typical (and, popular) Web-service. Thus an instance of MediaWiki[14] (version 1.28), as well as a MySQL server and the `memcached` cache daemon, were installed on each of the 48 application instances. The *wikiloader* tool from [194], and a dump of the database of the English version of Wikipedia from [195], were used to populate the MySQL databases, resulting in each instance containing an individual replica of the English Wikipedia.

Traffic and Workload Patterns

A traffic generator, able to replay a MediaWiki access trace and to record response times was developed, and experiments were run using 24 hours of traces from [195]. These traces correspond to 10% of all queries received by Wikipedia during this timeframe, from among which only traffic to the English Wikipedia was extracted and used for the experiment.

A first experiment was to size the server farm, *i.e.,* to identify the smallest number of application instances necessary to be able to serve queries while exhibiting reasonable response times. With 28 instances, the median response time during peak hours is smaller than 400 ms: the remainder of this section will assume this size for the server farm.

Connection Acceptance Policies Tested

Given the superior performance of $\mathbf{SR_4}$ and $\mathbf{SR_{dyn}}$ in the experiments from section 7.6.2, the 24-hour trace was replayed against both $\mathbf{SR_4}$ and $\mathbf{SR_{dyn}}$, and client-side response times were collected. As a baseline, the trace was also replayed against the reference **SC** policy.

The experiment allowed classifying queries into two groups: (i) requests for static pages, which are not CPU-intensive, and for which response times were of the order of a millisecond, and (ii) requests for wiki pages, that trigger `memcached` or MySQL and thus are more CPU-intensive. 6LB was found to offer only a small improvement over **SC** for static page response time (figure 7.24, top). However, the load times of wiki pages[15] exhibited interesting differences.

Figure 7.22 depicts the wiki page request rate and the median wiki page load time for the three tested policies during the 24h replay (data has been binned in 10 minutes slots). It can be observed that at the off-peak period around 8:00 UTC, when the system was lightly loaded and subject to a

[14] `https://www.mediawiki.org/wiki/Download`
[15] Those pages were identifiable by the string `/wiki/index.php/` in their URL.

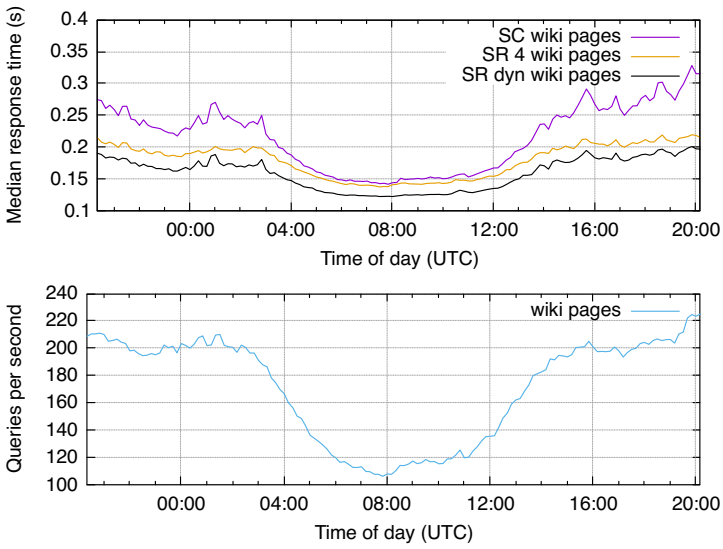


Figure 7.22 – 6LB Wikipedia replay: query rate and median wikipage load time

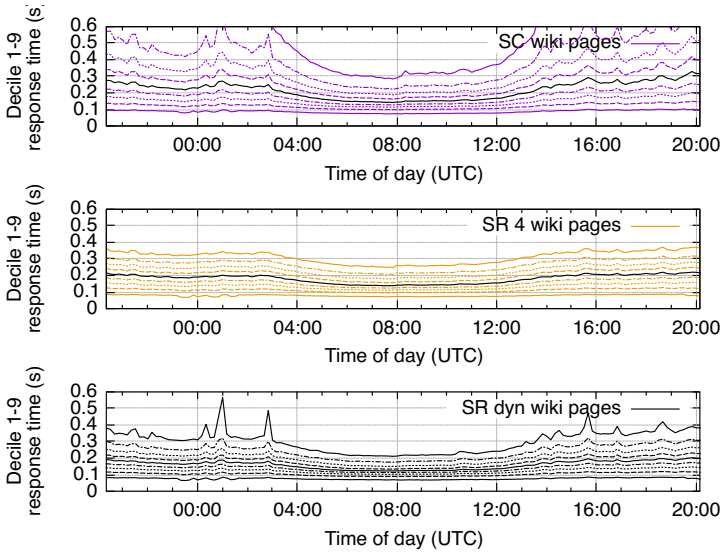


Figure 7.23 – 6LB Wikipedia replay: decile $1, \ldots, 9$ of wikipage load time

request rate of around 110 pages per second, **SC** and $\mathbf{SR_4}$ yielded similar performance, and $\mathbf{SR_{dyn}}$ exhibited even lower response times. As the request rate increases, using the application-unaware **SC** policy yielded notably increased page load times – whereas when using $\mathbf{SR_4}$ or $\mathbf{SR_{dyn}}$, a comparably much smaller increase in page load times incurred.

Experimental Results

To understand the response time variability over 24 hours, figure 7.23 depicts deciles 1-9 of the wiki page load time distribution, for each 10 minutes bin. Again, $\mathbf{SR_4}$ and $\mathbf{SR_{dyn}}$ show less variability under higher loads than does **SC**. Among $\mathbf{SR_4}$ and $\mathbf{SR_{dyn}}$, the latter has the lower variability under lighter loads, but is outperformed under higher loads.

Finally, as an indicator of "global good behavior", figure 7.24 (bottom) depicts the CDF of the wiki page load times over the whole day. Overall, the median response time went from 0.22 s with **SC** to 0.18 s with $\mathbf{SR_4}$ and 0.16 s with $\mathbf{SR_{dyn}}$. Furthermore, the tail of the distribution is steeper when using 6LB, with the 90-th percentile going from 0.67 s with **SC** to 0.32 s with $\mathbf{SR_4}$ and 0.31 s with $\mathbf{SR_{dyn}}$.

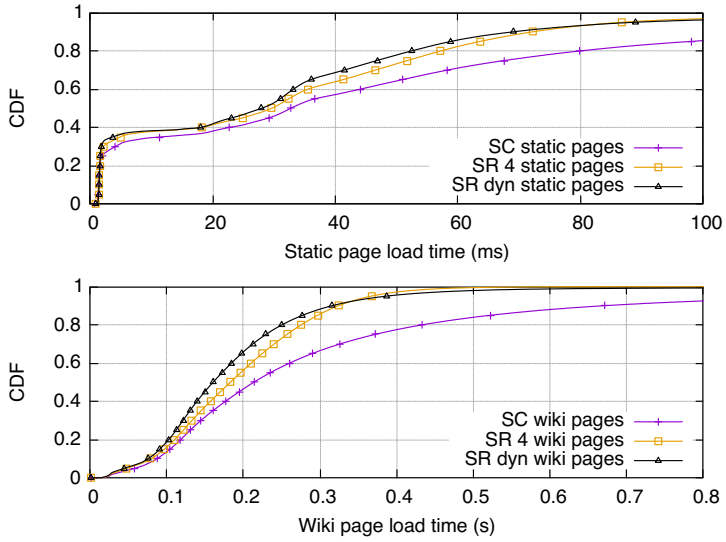


Figure 7.24 – 6LB Wikipedia replay: CDF of page load time over the 24 hours

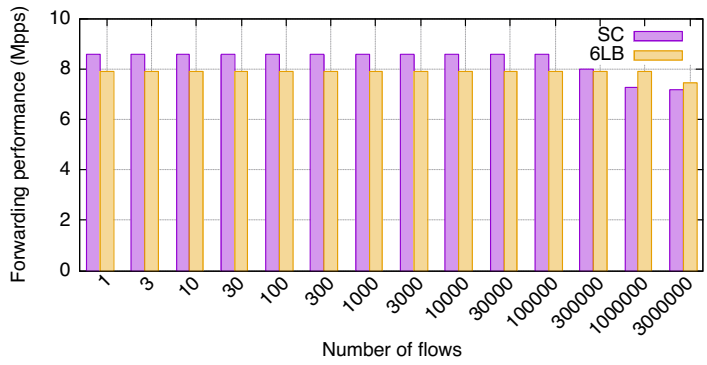


Figure 7.25 – Upstream packet forwarding rate evaluation: 6LB vs single-choice consistent-hashing, using a single CPU core.

7.6.4 Throughput Evaluation

The advantages provided by 6LB in load-balancing fairness come at the cost of some overhead as compared to single-choice load-balancing approaches, notably due to maintaining flow state and performing IPv6 SR header insertion. To understand the impact of 6LB in terms of CPU overhead, the packet-forwarding performance of the VPP implementation introduced in this chapter is evaluated. Maglev [162] with GRE encapsulation has also been implemented as a VPP plugin, and serves as a reference point. Evaluation was conducted on a single core of a machine running an Intel E5-2667 CPU at 3.2 GHz, with an Intel X710 10 Gbps NIC. The load-balancer was manually initialized to install a pre-determined number of flow entries, and a packet generator (sitting on another machine on the same Ethernet link) was set to send TCP ACK packets corresponding to these flows, at line rate. Packets were set to return to the packet generator, and the number of packets effectively forwarded by 6LB was recorded – allowing to determine the maximum forwarding capability of the implementation, for upstream traffic. ACK packets were used rather than SYN, as they are expected to represent the majority of the upstream traffic.

Figure 7.25 depicts the achievable forwarding rate (in millions of packets per second, Mpps), as a function of the number of flow entries installed. Two main results are to be noted. First, the kernel bypass and vectorization capabilities of VPP make it very efficient for load-balancing (be it single-choice or 6LB), with a raw forwarding capability of around 8 Mpps with one CPU core – whereas [162] reports 2.7 Mpps for the kernel bypass implementation of Maglev, and 0.5 Mpps without without kernel bypass. When the number of flows reaches approximatively 10^5, the performance of both implementations degrades, as the flow table cannot reside entirely in the CPU cache.

Second, it can be observed that 6LB incurs only 8% CPU overhead as compared to the load-balancer reference implementation. This overhead can be explained by the greater complexity of the per-packet operations, and the fact that the hash-table for flow state needs to handle collisions – whereas the one from the reference load-balancing plugin does not. Yet, this CPU overhead remains relatively negligible, and the additional 8% resources that might need to be deployed to use 6LB should be largely compensated by the fact that less application instances need be deployed, due to the greater fairness induced (as shown in section 7.6.2).

7.7 Summary of Results

This chapter has introduced 6LB, an innovative network service offering flexible, scalable, reliable, distributed, application-aware, but at the same time application-agnostic and application-protocol-agnostic, load balancing.

This is accomplished by an architecture in which (i) load-balancers use an extended consistent hashing algorithm to map incoming flows onto a set of candidate application instances, (ii) to offer – not impose – these network flows to the candidate application instances, leaving them the decision of whether or not to accept a flow. Once an application instance has accepted a flow (iii) data packets of no interest to the load-balancer are sent directly from the application instance to the client. When a network flow is reassigned to another load balancer (*e.g.,* if a load balancer is added to or removed from the system), this will be detected, and in-band signaling will reestablish the necessary state in this new load-balancer for continued operation, ensuring (iv) that a traffic flow between a client and an application instance becomes pinned to that application instance, regardless of changes to the load balancing infrastructure. The use of Segment Routing, specifically SR Functions, allow defining and implementing this as a network service, *i.e.,* entirely below the application layer.

This chapter has also introduced a simple two-choice random assignment policy (motivated by the concept of *power of two choices*), combined with a static or dynamic query acceptance policy. These policies were compared to a naïve one-choice random query dispatch policy, by way of an analytical model, as well as an evaluation on a 48-servers deployment. Evaluation of those policies, conducted using a simulated Poisson workload as well as on a Wikipedia replica, shows that 6LB is able to better spread the load between all application instances than single-choice consistent-hashing load-balancers. Evaluation of the packet-forwarding performance of the implementation shows that these benefits are attained at a negligible cost in terms of CPU overhead.

Results from this chapter have been published in [83, 84]. A subsequent study on how to use the data-plane of 6LB to improve cache hit-rates in Content Distribution Networks (CDNs), by performing popularity estimation in the data-plane, has been published in [196].

Chapter 8

Stateless Load-Aware Load Balancing in P4

In data-center and cloud architectures, as was detailed in chapter 1, workload are virtualized, with applications replicated among multiple application instances, each capable of independently serving incoming queries [2,3,197]. An important functional part in these architectures is the load-balancer (LB), dispatching incoming queries amongst application instances. To make the load-balancer "invisible", a Virtual IP Address (VIP), shared by all application instances providing the same service, is advertised to the Internet in place of the address of the load-balancer, requiring the load-balancer to provide per-connection consistency (PCC), *i.e.,* ensuring that traffic from a connection (typically identified by its network 5-tuple: source & destination addresses, L4 protocol, source & destination port) is *always* directed to the same application instance. *Naïve* load balancing use Equal Cost Multi-Path (ECMP) [161] to map connections to application instances, using a hash function and a modulo operation.

8.1 Statement of Purpose

A drawback of using ECMP for load-balancing is the lack of resiliency to changes to the application instance set, which causes the modulus in the ECMP operation to change and most connections to be redistributed across application instances, thus breaking PCC and causing connection resets. *Consistent hashing* [166–168] attempts to address this, by providing a more resilient mapping of connections across application instances [162, 165] through maintaining an intermediate table which, with high probability, yields a persistent mapping of the 5-tuple space to the set of application instances, even when faced with changes in the application instance set. Maglev [162] uses consistent hashing with per-connection state to maximize the probability of PCC: a connection breaks only when **both** (i) per-connection state is removed (if memory is exhausted, for instance due to a denial-of-service attack – or if traffic is rebalanced to a new LB instance) **and** (ii) consistent-hashing changes the mapping of the connection to a different application instance (if there is a change in the set of application instances, which should affect only a small number of connections).

The pseudo-random nature of consistent hashing assigns queries to application instances *regardless* of their actual load state [162]. While this does not pose any problem for non-CPU-intensive applications (*e.g.,* serving static Web pages), performance may degrade for CPU-intensive applications (*e.g.,* data processing), for which the number of concurrently served queries per application instance must be minimized. Assigning queries to the least loaded from among *two* randomly chosen application instances (rather than to *one* randomly chosen application instance) was shown in [164] to improve load-balancing fairness. Based on this, 6LB, introduced in chapter 7, combines a Maglev-like consistent hashing with assigning connections to set of two application instances, which decide amongst themselves which will accept the connection. This is achieved by forwarding connection request (SYN) packets using Segment Routing (introduced in section 1.2.1) and then by maintaining state in the LB as to which instance has accepted the connection.

The need to maintain per-flow connection-state make Maglev and 6LB difficult to implement on programmable hardware devices, whereas it is known that hardware-based load-balancers offer

a potential performance benefit [169, 198, 199]. Beamer [199] circumvents this state requirement, by calling on assistance from the network stack of application instances to maintain PCC. When the set of application instances changes, and consistent hashing maps a flow to a different application instance, Beamer directs packets from that flow to that *new* application instance, while also embedding the address of the previously used application instance in the packet header. This allows the *new* application instance, in case it does not have connection state for a received packet, to forward it to the previous instance – somewhat similarly to the recovery mechanism of 6LB introduced in section 7.4.3.

A shared requirement of [84, 199] is that an application instance is able to direct packets to a second application instance, when needed. For 6LB, only connection establishment packets (or first connection recovery packets) are proposed to two application instances, allowing load sharing – at the expense of state in the LB for forwarding subsequent packets directly to the correct application instance, using the stickiness mechanism of section 7.4. Beamer offers packets to two application instances only when there is a change in consistent hashing – but then, does so for all packets in a flow, to avoid keeping state in the LB.

Thus the question: *is it possible to provide a stateless load-balancer that dispatches queries according to the state of the applications?* This requires the LB to be able to (i) send connection requests through a chain of "candidate" application instances for local connection acceptance decisions, and (ii) statelessly direct packets in an established flow to the one application instance which accepted the connection request.

As a possible solution, this chapter introduces SHELL, an application-agnostic, application-load-aware, stateless load-balancer, which (i) proposes new connections to a *set* of pseudo-randomly-chosen application instances, each making a local acceptance decision, and (ii) "marks" subsequent packets in a flow so as to allow the load-balancer to direct them to the appropriate application instance, without requiring the load-balancer to maintain per-connection-state.

The statelessness of the load-balancer makes it a candidate for an implementation in a hardware platform. Thus, this chapter proposes a prototype P4 [200] implementation of SHELL targeting the NetFPGA SUME [201] platform using the P4-NetFPGA [202] framework, as well as an extensive performance evaluation of the P4 implementation and of the stable hashing algorithm.

8.1.1 Chapter Outline

The remainder of this chapter is organized as follows. Section 8.2 provides an overview of SHELL, with section 8.3 detailing key design aspects. The P4-NetFPGA implementation of the load-balancer is detailed in section 8.4, followed by a performance evaluation in section 8.5. Resiliency of consistent hashing is evaluated in section 8.6, before section 8.7 concludes this chapter.

8.2 Overview

SHELL consists of 3 main components: a control plane, a P4-based load-balancing data-plane, and a server agent.

The control-plane constructs two tables (see section 8.3.1): (i) a *consistent hashing* table, used to direct new connection request packets (*e.g.,* TCP SYNs) to a set of candidate application instances, which improves *fairness*, and (ii) a *choice history* table, for directing subsequent packets in a flow (*e.g.,* TCP ACKs), which improves *resiliency*.

The P4 load-balancer uses 5-tuple hashing to map each new connection request to a list of candidate application instances from the *consistent hashing* table provided by the control-plane. Segment Routing is then used to direct such packets through the selected list of application instances, until one accepts the connection (note that the last application instance in the set must always accept), as in 6LB (chapter 7). Then, $\mathcal{C}_i$, the position in the list of the application instance which accepted the connection, is communicated back to the client (see section 8.3.2), which in turn includes it in all further packets from the client to the load-balancer. This enables the P4 load-balancer to send these packets directly to the application instance handling the connection. When a change in the set of application instances causes modifications to some of the *consistent hashing* buckets, changes are saved in a *history*. The P4 load-balancer then directs (using SR) subsequent packets to the current and previous application instances associated with the value $\mathcal{C}_i$ received in the packets.

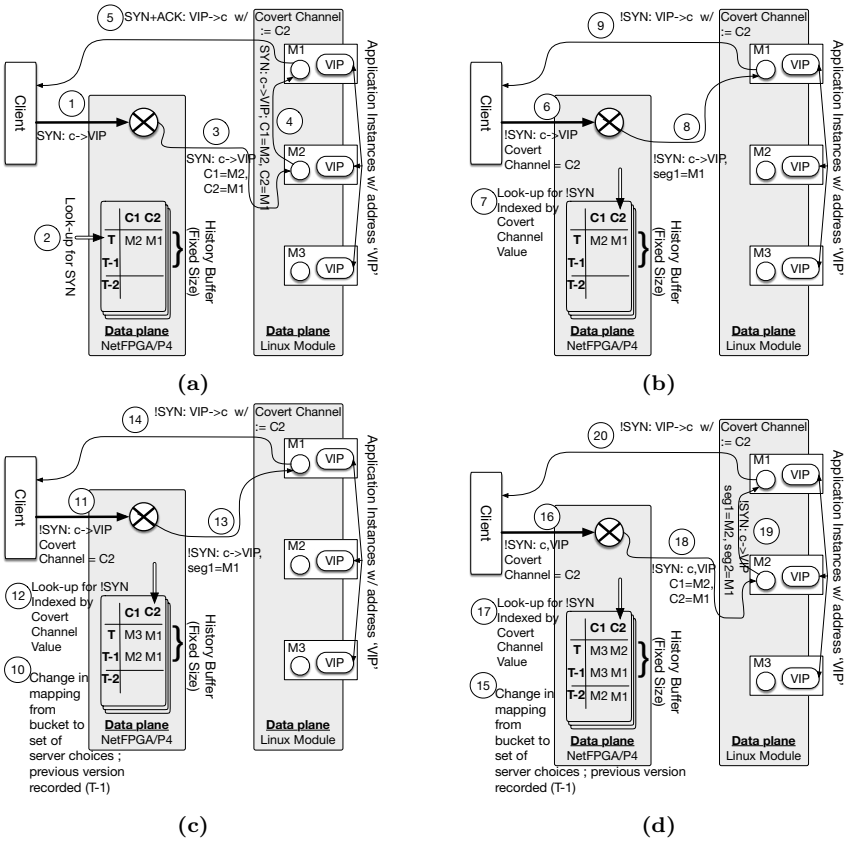


Figure 8.1 – SHELL overview

Finally, the server agent (i) accepts new connection requests or forwards them to the next candidate application instance, and (ii) forwards further packets until reaching the application instance that accepted the connection. This server agent is implemented as a Linux kernel module; the details hereof are out-of-scope for this chapter.

An exemple execution is illustrated in figure 8.1. (a) A SYN packet is directed through $c = 2$ candidate instances (c_1, c_2). In this example, the two instances are hosted on machines m_2, m_1, and the second candidate (hosted on m_1) accepts the connection. This choice is reported in the covert channel of the SYN-ACK packet. More precisely, the covert channel encodes a label $\mathcal{C}_2$ indicating that the second candidate accepted the connection (rather than encoding the full value m_1 of the address corresponding to this choice), thus carrying only one bit of information. (b) Subsequent packets from the client are directed by the LB to the correct machine m_1, by looking up the machine corresponding to the covert channel value $\mathcal{C}_2$ in a consistent hashing table. (c) Upon reconfiguration of the pool of instances, with high probability the assigned instance is not modified in the consistent hashing table, due to the resiliency property of consistent hashing. (d) A reconfiguration modifies the candidate list to $(c_1^t, c_2^t) = (m_3, m_2)$. SHELL then uses the history matrix to go through (the machines hosting) the previous instances that used to be the c_2 for this bucket – in this example, $(c_2^t, c_2^{t-1}) = (m_2, m_1)$.

8.3 Description

In this section, c is the number of candidate application instances through which connection requests are directed, B the number of buckets used in consistent hashing, and h the *"depth"* of the consistent hashing *"history matrix"* maintained for a bucket $t[b]$ (an example of $t[b]$ for a bucket b is shown in table 8.1). Finally, c covert channel labels $\mathcal{C}_1, \ldots, \mathcal{C}_c$ are defined, where label $\mathcal{C}_i$ indicates that "at the time of connection establishment, this connection was accepted by the i-th application instance in the SR list embedded in the packet".

The behavior of the P4 load-balancer and of the application instance depends on which packet

Algorithm 10 Consistent hashing history table construction

▷ *update version numbers*
for $b \in \{0, \ldots, B-1\}, i \in \{h-1, h-2, \ldots, 1\}, j \in \{0, \ldots, c-1\}$ **do**
  $t[b][i][j] \leftarrow t[b][i-1][j]$
end for
▷ *build new consistent hashing table*
for $b \in \{0, \ldots, B-1\}, j \in \{0, \ldots, c-1\}$ **do**
  $t[b][0][j] \leftarrow consistentHashing(b, j)$
  ▷ *use algorithm 9*
end for
▷ *remove duplicates*
for $b \in \{0, \ldots, B-1\}, j \in \{0, \ldots, c-1\}$ **do**
  if $t[b][0][j] = t[b][i][j]$ for some $i \neq 0$ **then**
    delete $t[b][i][j]$ and shift $t[b][i+1, \ldots, h-1][j]$ upwards
  end if
end for

	Choice 1	...	Choice c
Epoch t	m_3	...	m_4
Epoch $t-1$	m_2	...	m_{13}
⋮	⋮	⋮	⋮
Epoch $t-h+1$	m_{13}	...	m_{10}

Table 8.1 – Example entry of the history matrix, for a given bucket

is received:

1. **A connection request (TCP SYN) received at the LB** is hashed, on its 5-tuple, into a bucket with index b. The first row of the history matrix for b, $t[b][0][:]$, is then used for generating an SRv6 header with a segment list of $(t[b][0][0], \ldots, t[b][0][c-1], v)$, where v is the original VIP included in the packet. In the example from table 8.1, the inserted SR header is $(m_3, \ldots, m_4, v)$.

2. **A connection request (TCP SYN) received at an application instance** is processed by the corresponding server agent. The agent examines the local state of the application (*e.g.,* its CPU load), and determines to either accept the request, or to forward the packet to the next segment in the SR header, using *e.g.,* algorithm 7 or algorithm 8 from chapter 7. Note that the last candidate in the list must always accept the connection.

   A server agent having accepted a connection will record, for the connection lifetime, its own index $\mathcal{C}_i \in \{\mathcal{C}_1, \ldots, \mathcal{C}_c\}$ in the segment list received with the connection request, so as to be able to signal this to the client, through the covert channel. In the example from table 8.1 where the inserted SR header is $(m_3, \ldots, m_4, v)$, if the first candidate (m_3) accepts the connection, the corresponding server agent will record $\mathcal{C}_1$ as a covert channel label.

3. **When an application instance emits a packet,** the corresponding server agent will embed the recorded label $\mathcal{C}_i$ for that connection in the TCP SYN-ACK (and in *all future outgoing packets for that connection*), using a covert channel – how this is accomplished is discussed in section 8.3.2.

4. **All subsequent (*i.e.,* TCP non-SYN) packets emitted by the client** will automatically encode $\mathcal{C}_i$, since the covert channel is automatically reflected by the network stack of the client.

5. **All subsequent (*i.e.,* TCP non-SYN) packets received at the LB** are hashed, on their 5-tuple, identifying a bucket b. The LB also extracts the label $\mathcal{C}_i$ inserted in the covert channel by the client. The i-th colum of $t[b]$ is then used to generate an SRv6 header with segment list $(t[b][0][i-1], \ldots, t[b][h-1][i-1], v)$.

   In the example from table 8.1, if the value in the covert channel is $\mathcal{C}_1$, the inserted SR header is $(m_3, m_2, \ldots, m_{13}, v)$. This allows the packet to reach m_3, where it is the most likely that the

connection is handled. However, if the history matrix has been updated and the connection had been opened before that update, it is possible that the flow is not handled by m_3. When this is the case, since $m_2, \ldots, m_{13}$ are also included in the SR header, the packet is directed through the previous instances corresponding to that bucket and that choice index $\mathcal{C}_1$, until reaching the correct one.

6. **All subsequent (*i.e.,* TCP non-SYN) packets received at the application instance** will be examined, and if corresponding connection-state is found, will be processed locally. Otherwise, such packets are forwarded to the next segment, allowing to try an "older" instance which used to map to the same bucket and label $\mathcal{C}_i$. In the unlikely event that there is no next segment, the packet is dropped – this corresponds to the case where the history was not long enough to include the application instance that had accept the connection in the first place.

8.3.1 History Matrix Computation

First, the same algorithm as in 6LB (algorithm 9) is used to generate, for each bucket b, a *candidate list* of application instances $\ell = (m_1, \ldots, m_c)$. This list is recorded as the first row of the history matrix for the bucket: $t[b][0][:] = \ell$. When the set of application instances is modified, the history matrix needs to be modified. Algorithm 10 describes the mechanism used to generate the history matrix when this occurs:

- First, all entries in the history matrix are offset by one, *i.e.,* $\forall b, t[b][i+1][:] \leftarrow t[b][i][:]$.
- Then, the new mapping of choice index to candidate application instances is calculated for each bucket b, again using algorithm 9.
- By design, and as depicted in figure 7.5 and [162, figure 12], consistent hashing will leave most bucket entries unmodified after reconfigurations of the application pool, *i.e.,* $t[b][0][:] = t[b][1][:]$ for most b. Therefore, duplicate history entries are removed (last step of algorithm 10), so as to avoid repetitions in the created SR headers, and to span a potentially longer history of application instances having occupied a bucket.

8.3.2 Possible covert channels

The server-state-aware and stateless load-balancing approach described in this chapter relies on the application instance being able to instruct clients to include $\mathcal{C}_i$ (the position of the application instance in the segment list at time of connection establishment) in all packets subsequent to the connection request. This label $\mathcal{C}_i$ is then used by the load balancer to direct these packets to the appropriate application instance.

If it is possible to modify the client networking stack so as to cooperate with the load-balancing architecture, a simple option would be to embed the full identifier (rather than a label $\mathcal{C}_i$) of the instance having accepted the connection in packets sent to the client, then reflect this identifier in the packets sent by the client. This can be achieved with transport protocols such as QUIC [146], which embed a connection identifier chosen by servers and reflected by clients.

However, for TCP connections, it is necessary to convey $\mathcal{C}_i$ from the application instance to the client, and make the client relay $\mathcal{C}_i$ in subsequent packets, without client-side modification[1]. This requires using a *covert channel from the application instance, through the client, and to the load-balancer*, which the client neither inspects nor interferes with. Approaches accomplishing this include:

- **TCP sequence numbers**, which are reflected by endpoints in the acknowledgement field. Since the initial sequence number is chosen by the server, this allows implementing the covert channel through the high-order $\lceil \log_2 c \rceil$ bits of the sequence number. Note that this method fails for connections whose length outrun $1/c$ of the sequence number space (*i.e.,* $2^{32}/c$ bytes), because in this case the high-order bits will be modified in the middle of the

[1] The rationale for embedding a label $\mathcal{C}_i$ (rather than the full application identifier) in the covert channel, is that it is a more compact representation, requiring only $\lceil \log_2 c \rceil$ bits. In practice, if $c = 2$ (as is usual from the "power of two choices" [164]), only 1 bit is required. If one were to encode the full instance identifier in the covert channel, this would require $\lceil \log_2 |\mathcal{M}| \rceil$, where $\mathcal{M}$ is the set of machines capable of hosting those instances.

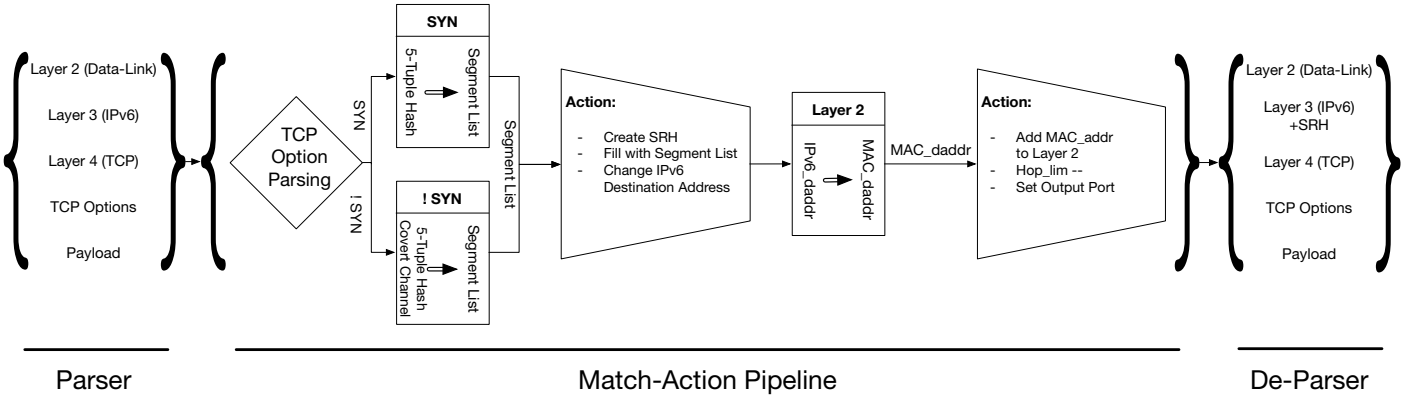


Figure 8.2 – SHELL P4 data-plane overview

```
header tcp_opts_24_t {
  bit<32> nopnoptypelength1; bit<63> value1; bit<1> potential_cov_channel1;
  bit<32> nopnoptypelength2; bit<63> value2; bit<1> potential_cov_channel2;
}
state parse_tcp_opts_24 {
  packet.extract(hdr.tcp_opts_24);
  transition select(hdr.tcp_opts_24.nopnoptypelength1) {
    32w0x0101080a : set_covert_channel_24_1; /* NOP NOP TS (type=08, length=0a) */
    32w0x0101050a : parse_tcp_opts_24_2; /* NOP NOP SACK10 (type=05, length=0a) */
    default: accept;
  }
}
state set_covert_channel_24_1 {
  user_metadata.covert_channel = hdr.tcp_opts_24.potential_cov_channel1;
  transition accept;
}
state parse_tcp_opts_24_2 {
  transition select(hdr.tcp_opts_24.nopnoptypelength2) {
    32w0x0101080a : set_covert_channel_24_2; /* NOP NOP TS (type=08, length=0a) */
    default: accept;
   }
}
state set_covert_channel_24_2 {
  user_metadata.covert_channel = hdr.tcp_opts_24.potential_cov_channel2;
  transition accept;
}
```

Figure 8.3 – Example TCP TLV parsing in the P4 LB, for $d_{\text{off}} = 11$.

connection. In this case, it would be necessary to implement a mechanism to detect when the high-order bits are about to wrap, as well as a flow table to record flow state when this happens. Alternatively to recording flow state, the LB could insert an SR list comprising all $c \times h$ segments of the history matrix to perform a recovery through all candidates, but this would potentially have a high network overhead.

- **TCP timestamps**, which are also reflected by TCP endpoints. The covert channel can be carried in the $\lceil \log_2 c \rceil$ low-order bits of the TCP timestamp, as is done in [203] – and according to [204, 205] this is resilient to middlebox processing. Modifying the timestamp by at most $c-1$ units ($c=2$ usually [164]) will have a negligible effect on RTT estimation.

To avoid using a flow table, SHELL uses **TCP timestamps** as covert channel for TCP connections: the server agent of the accepting application instance encodes its index $\mathcal{C}_i$ in the low-order bits of the TCP timestamp – and the LB inspects TCP timestamp sent by clients.

8.4 P4 Load-Balancer Implementation

This section describes the P4 implementation of the data-plane introduced in section 8.3.

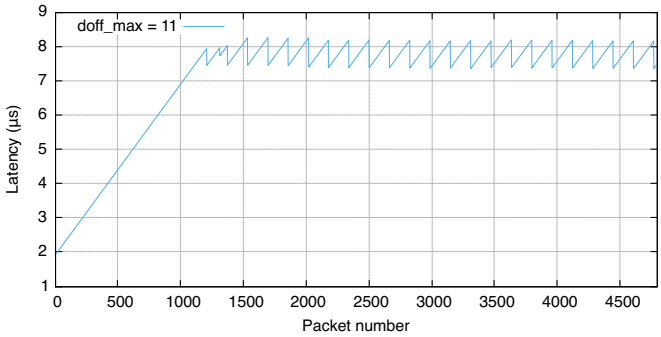


Figure 8.4 – SHELL P4 dataplane evaluation: per-packet latency for a burst of 4800 packets ($c = 2, h = 2, d_{\text{off}}^{max} = 11$)

Throughput (Mpps)	59.8
Worst-case latency (μs)	8.96

Table 8.2 – SHELL P4-NetFPGA dataplane performance

8.4.1 Data Plane

The workflow of the P4 implementation of the LB data-plane of SHELL is illustrated in figure 8.2. It uses three match-action tables, corresponding to (i) segment lists to be inserted into SYN packets, as given by the first row of the history matrix, (ii) segment lists to be inserted in non-SYN packets, as given by columns of the history matrix, and (iii) a Layer-2 lookup for output packets.

Processing a packet starts with parsing its headers. If they do not match the expected format (*e.g.,* the packet uses UDP), the packet is dropped; otherwise, they are processed by the match-action pipeline. Both (i) the 5-tuple hash of the packet, computed by way of an external function from the P4-NetFPGA framework, and, (ii) in case of a non-SYN packet, the covert channel value as found in the TCP timestamp field (whose parsing is described below), are used as keys to the match tables, in order to access the corresponding segment list. The segment list is then fed to an action, which builds the SR header. To complete the match-action pipeline, a match is performed on the newly-added destination address of the packet (*i.e.,* the first SR segment), which calls an action making the packet egress through the correct interface. Finally, the de-parsing stage emits the packet with the newly-built headers.

The TCP timestamp is embedded in an option of the TCP header, and TCP options are formatted as Type-Length-Values (TLVs) [17] – which makes it impossible to know in advance where to peek into the packet so as to retrieve its TCP timestamp. For want of a lookahead function and of variable-length header support in the P4 compiler, only a set of "reasonable options" is parsed (specifically, SACKs and timestamps, *i.e.,* those that can be found in non-SYN packets according to [204]). Depending on the length of the TCP header found in the "data offset" (denoted by d_{off}) field, an option header with the identified possible combinations of TLVs is parsed, corresponding to lengths $d_{\text{off}} \in \{8, 11, 13, 15\}$. Figure 8.3 depicts an example of such a parsing, when $d_{\text{off}} = 11$. The bits of the covert channel are then stored into a meta-data field[2]. A parameter of interest to the performance of the data-plane is the maximum size of the parsed TCP header d_{off}^{max}: its influence on the performance is evaluated in section 8.5.

8.5 P4-LB Implementation Performance

A key element to the performance of SHELL is the per-packet latency incurred in the P4-dataplane of the LB. It depends primarily on two factors: (i) the latency incurring when receiving a packet over an ingress interface, inserting an SR header, and transmitting this (now larger)

[2]To ensure compatibility with clients not using TCP timestamps, if a SYN packet is missing the TCP timestamp option (detected by $d_{\text{off}} \leq 8$), an SR header with only one candidate will be inserted. Lack of a TCP timestamp in non-SYN packets is then interpreted as a covert channel value of $\mathcal{C}_1$.

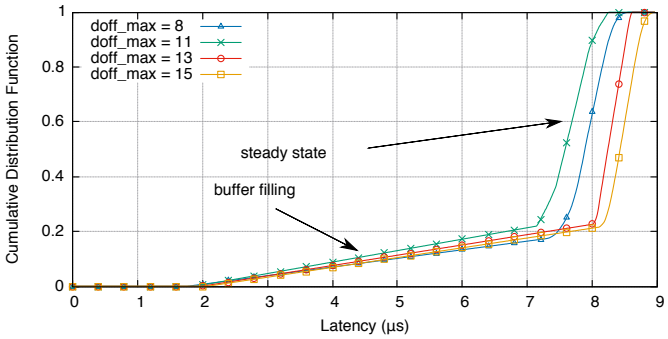


Figure 8.5 – SHELL P4 dataplane evaluation: distribution of per-packet latency for a burst of 4800 packets ($c = 2, h = 2$, different d_{off}^{max})

Max TCP size	LUT	LUTRAM	FF	BRAM
$d_{\text{off}}^{max} = 8$	36.9%	19.4%	33.3%	59.3%
$d_{\text{off}}^{max} = 11$	40.1%	22.0%	36.4%	63.2%
$d_{\text{off}}^{max} = 13$	43.8%	24.9%	40.2%	67.7%
$d_{\text{off}}^{max} = 15$	48.7%	28.6%	45.8%	74.1%

Table 8.3 – SHELL P4-NetFPGA dataplane resource usage on a NetFPGA-SUME. (LUT: Look-Up Tables; LUTRAM: Look-Up Tables used as RAM; FF: registers (Flip-Flops); BRAM: Block RAM.)

packet over an egress interface (section 8.4.1), and (ii) the latency incurring from extracting $\mathcal{C}_i$ from within the TCP timestamp (section 8.3.2).

These factors are evaluated using the P4-NetFPGA framework and the Xilinx Vivado software suite, simulating packets going through one interface of a 10G NetFPGA-SUME, and are documented in this section.

8.5.1 Effect of SR Header Insertion on Latency

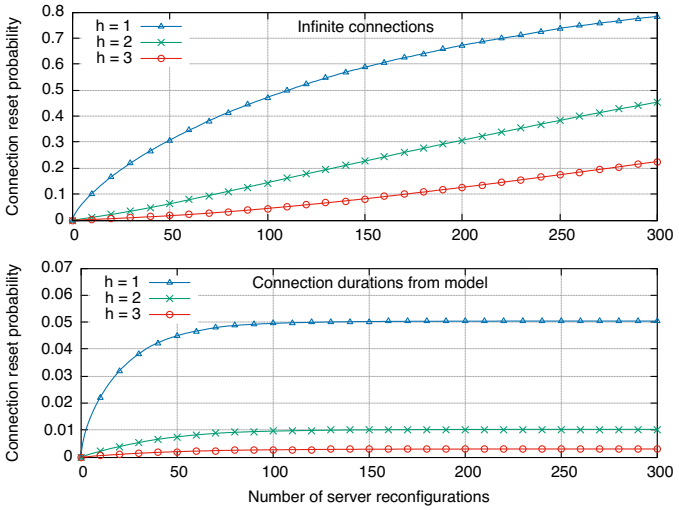
For the purpose of the simulations, $c = 2$ choices were used for SYN packets, and a history of $h = 2$ was used for ACK packets, thus SR headers with 3 segments were inserted, increasing the packet size by 56 bytes between ingress and egress. Feeding ingress packets at maximum line-speed fills the egress interface queue over time, thus progressively increasing packet forwarding latency – as depicted in figure 8.4, showing stable results from above 1500 packets. When the egress buffer is empty, the latency is 2.1 μs, and when the buffer is full the latency oscillates between 8.2 μs and 9.0 μs. Thus, for subsequent simulations, batches of 4800 packets are injected at line-rate on the ingress interface.

The batch of packets consists of a mixture of SYN packets, ACK packets with odd TCP timestamps and ACK packets with even TCP timestamps. Table 8.2 reports the throughput and worst-case latency obtained: SHELL, comparable to Beamer [199], can sustain 60 Mpps, *i.e.,* 22× as much as what is reported in [162] for the single-core software implementations of Maglev, while also providing application instance load-awareness.

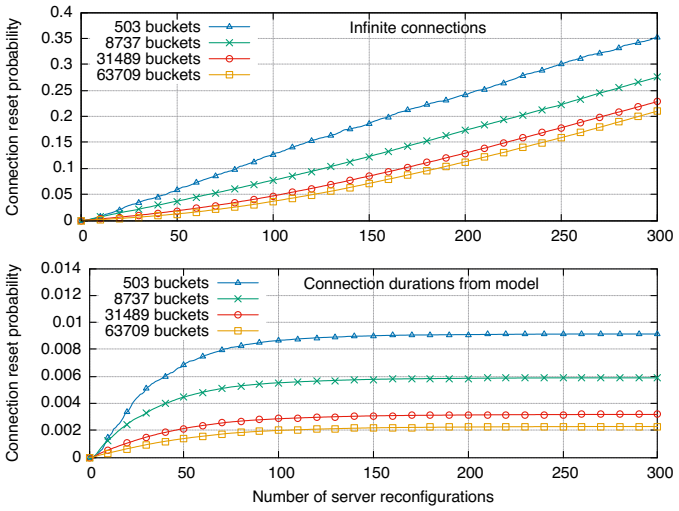
8.5.2 Effect of TCP Parsing on Latency and FPGA Resources

As explained in section 8.4, the TCP timestamp option is parsed by matching a predefined set of option headers, thus the greater d_{off}^{max} (maximum admissible size of parsed TCP headers), the more program branches – and the greater the latency. This is evaluated by testing different values of $d_{\text{off}}^{max} \in \{8, 11, 13, 15\}$, and depicted in figure 8.5. The latency varies from 1.8 μs ($d_{\text{off}}^{max} = 8$) to 2.1 μs ($d_{\text{off}}^{max} = 15$) for an empty egress queue, and its average after the egress queue has filled up goes from 7.7 μs ($d_{\text{off}}^{max} = 11$) to 8.5 μs ($d_{\text{off}}^{max} = 15$).

This allows noting that, in a controlled environment (*e.g.,* a data-center, where SHELL typically would be deployed), where a strict set of TCP options can be enforced, it is possible to trade off parsing safety against reduced latency. Conversely, parsing more potential options increases the amount of logic consumed on the FPGA. As reported in table 8.3, which depicts the resource usage



(a) $B = 35591$ buckets, different history depths h. Top: infinite connections. Bottom: connections according to model.



(b) $h = 3$, different number of buckets B. Top: infinite connections. Bottom: connections according to model.

Figure 8.6 – SHELL consistent hashing evaluation: probability for a connection started at time $t = 0$ to be reset after n backend reconfigurations. 500 application instances, $c = 2$ choices. Backend reconfiguration rate model from [198, 206].

on the FGPA for different values of d_{off}^{max}, this can increase LUT (Look-Up Tables, implementing the FPGA logic) usage by up to one third.

8.6 Consistent Hashing Resiliency

The resiliency of the consistent hashing algorithm detailed in 8.3.1 is evaluated by way of simulations. Long-lived connections are particularly vulnerable to application instance reconfigurations – thus, as a *worst-case* scenario, simulations using *infinite-length connections*, and subject to successive application instance removal/insertions, are performed; each simulation is repeated 5 times.

For a random connection, assuming that it was uniformly drawn from among the $B \times c$ buckets and choices possible, the probability that it is reset[3] after n application instance reconfigurations is depicted in figure 8.6a (top), which shows the impact of the history depth h, and figure 8.6b

[3] A connection assigned to a bucket/choice pair is deemed reset after n reconfigurations if the application instance to which it was initially assigned no longer appears in the corresponding history table.

(top), which shows the impact of the number of buckets B.

In realistic scenarios, application instance reconfigurations are expected to be rare, and connections are unlikely to last for more than a few application instance reconfigurations – at most. The bottom graphs of figures 8.6a and 8.6b depict the connection reset probability using the connection duration distribution model from [206, Figure 7.a] (where connections have a median duration of 365 ms), and with the application instance reconfiguration rate distribution taken as the *highest 1% rates over a month for 100 clusters* from [198, Figure 2] (in this worst-case distribution, reconfigurations have a median rate of 13.5 min^{-1}).

In these conditions, less than 1% of the connections were lost when using SHELL with a non-void history ($h > 1$). In comparison, without history (*i.e.,* $h = 1$, as would be the case in Maglev and 6LB), more than 5% of the connections were lost.

8.7 Summary of Results

This chapter has introduced SHELL, an application-agnostic load-balancing architecture which combines application load awareness (by using a *power-of-choices* scheme upon connection establishment), statelessness (by using a *covert channel* to indicate which of the candidates had accepted the connection), and resiliency (by using *consistent hashing* and *versioning*). Being stateless makes SHELL suitable for a hardware implementation, as demonstrated in this chapter through prototype development using the P4-NetFPGA framework. An evaluation of throughput and latency of this prototype implementation shows that the attainable performance is equal to that of other hardware implementations, while providing application-awareness and therefore improving load-balancing fairness. Further, simulation of the consistent hashing resiliency shows that the number of long-lived connections dropped, even in worst-case scenarios, is negligible.

Results from this chapter have been published in [85].

Chapter 9

Joint Auto-Scaling and Load-Balancing with Segment Routing

Virtualization and cloud architectures, wherein different tenants share computing resources to deploy their workloads, reduce the possible granularity of task allocation in data centers [3]. To optimize cost and energy, applications can be (i) replicated among multiple instances running in containers or virtual machines (VMs) [1, 2], and (ii) automatically scaled up or down in order to meet a given Service Level Agreement (SLA) [207]; this, to trade off cost and user *quality of experience*. Two functions serve this purpose: (i) a *load-balancer*, which dispatches queries onto identical instances, and (ii) an *autoscaler*, which monitors application instances to scale up or down the number of application replicas.

A main challenge for network load-balancers is to provide performance and resiliency, while taking application state into account. Some architectures, like Equal Cost Multi-Path (ECMP) [161] or Maglev [162], distribute flows among instances pseudo-randomly, allowing forwarding packets without terminating Layer-4 connections, and thus providing a high throughput. The use of consistent hashing (as in [162]) also allows for resiliency, in case an existing flow is handed over to another load-balancer. Such architectures, nonetheless, assign flows to instances regardless of their load state – but it has been demonstrated [164] that considering application load can greatly improve overall performance. Other load-balancing architectures do take application state into account, by terminating Layer-4 connections [185], and/or using centralized monitoring [172] – thus incurring a performance overhead and degrading resiliency.

Similarly, autoscalers use centralized monitoring, with an external agent gathering load metrics from all application instances and collapsing these into a decision [207]. This can therefore sometimes cause decisions to be made on out-of-date information. Furthermore, such agents typically collect external metrics (*e.g.,* CPU load of a VM as seen by the hypervisor), ignoring application-provided metrics that could possibly be more suitable to make scaling decisions.

9.1 Statement of Purpose

The purpose of this chapter is to introduce a unified architecture for load-balancing and autoscaling, and to analyze and quantify its characteristics. The proposed architecture supports application-load-aware load-balancing and autoscaling decisions, thus eliminating the need for centralized monitoring and decision-taking. To decrease operational overhead, the proposed architecture operates entirely within the network layer (Layer-3), removing the need from terminating or proxying network connections. Using the dataplane of 6LB (introduced in chapter 7), queries are directed with SR through several application instances, each able to make a local decision to accept or refuse the query, and the last one able to make autoscaling decisions. In sum, and as described in section 9.2, the local state of the applications is used to make **both** load-balancing decisions and autoscaling decisions. The contributions of this chapter are twofold:

1. An analytical model of the behavior of the proposed load-balancing policy for a chain of n servers is formulated. The model is solved by extending the Recursive Renewal Reward (RRR)

technique [208] to n-dimensional Markov chains, and numerical results are obtained when the state space is reasonably small.

2. The proposed load-balancing and autoscaling architecture is implemented within a virtual router (VPP [31]), and tested against traces of a real workload, allowing to confirm that the properties of the system hold in real environments.

9.1.1 Related Work

This section reviews the literature on load-balancing and autoscaling, before presenting a mathematical framework (the Recursive Renewal Reward technique) used throughout this chapter. Segment Routing has been introduced in section 1.2.1, and work related to load balancing has been reviewed in section 7.1.1.

Autoscaling

Methods to provide autoscaling have been classified by [207] as *reactive* and *proactive*. Reactive methods consist of regularly gathering measurements, and taking actions accordingly when thresholds are violated. For instance, in [209] up/downscaling is triggered when bounds on some observed metrics are violated; [210] has a similar approach with dynamic threshold adjustment. Such approaches incur an overhead due to the gathering of statistics, and a time gap between detection of violations and appropriate reaction.

Conversely, proactive approaches consist of anticipating changes and acting correspondingly. Machine learning techniques are used in [211] to classify workloads by their resource allocation preferences, and [212] uses control theory to track CPU usage and to allocate resources accordingly. While solving the issue of timeliness, proactive approaches suffer the need to collect statistics and perform centralized computations. This chapter avoids this by enabling applications instances to make local autoscaling decisions.

Recursive Renewal Reward

The Recursive Renewal Reward technique (RRR) [208] has been introduced as a means to extract metrics from a class of Markov chains. It consists of identifying a *home state* S_0 and a *metric of interest* $M(t)$ (the *reward*). Then, this allows evaluating the expected value of M by computing its average earning rate over a cycle from S_0 to itself, divided by the average duration of the cycle:

$$\mathbf{E}[M] = \frac{\mathbf{E}[\int_{S_0 \to S_0} M(t)\mathrm{d}t]}{\mathbf{E}[\int_{S_0 \to S_0} 1\mathrm{d}t]} \tag{9.1}$$

RRR applies to bi-dimensional Markov chains with a one-dimensional repeating pattern, in which case computing the average earning rate of the reward reduces to solving a finite number of equations, by restricting to the *border row* that generates the pattern. RRR has been applied to various problems, from vehicular networks [213] to query duplication in data centers [214].

9.1.2 Chapter Outline

The remainder of this chapter is organized as follows. Section 9.2 gives an overview of the architecture introduced in this chapter. Using RRR, an analytical model for the response time of the system is introduced for $n = 2$ servers in section 9.3, and extended for $n \geq 3$ servers in section 9.4. Numerical results are given in section 9.5, then results of experimentation on a realistic testbed are presented in section 9.6. Finally, section 9.7 concludes this chapter.

9.2 Joint Load-Balancing and Autoscaling

This section presents an overview of the architecture introduced in this chapter. Application instances are ordered into a chain, and an SLA on response times is decided by the data-center operator. A load-balancer dispatches requests through this chain with SR, using the dataplane of 6LB introduced in section 7.4. Each of the instances in the chain decides to accept the connection if this yields to *locally* satisfying the SLA, otherwise forwards it to the next instance in the chain, as illustrated in figure 7.3. This way, queries are served by the first instance of the chain able

Algorithm 11 Local Autoscaling at Last Instance

```
c_v ← 0    ▷ threshold violations
w ← 0    ▷ window size
ε_up ← 0.75    ▷ upscaling bound
ε_down ← 0.40    ▷ downscaling bound
t_0 ← time
for each connection establishment packet p do
    v ← p.lastSegment
    b ← number of busy threads for v
    w++
    if b ≥ c then
        c_v++
    end if
    if t − t_0 > 1 min then
        if c_v/w > ε_up then
            request upscaling
        else if c_v/w < ε_down then
            request downscaling
        end if
        c_v ← 0; w ← 0; t_0 ← time
    end if
    p.segmentsLeft ← 0
    p.dst ← v
    forward p to local workload v
end for
```

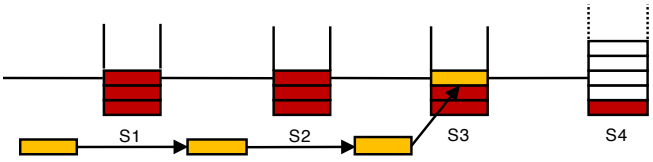


Figure 9.1 – First-available-instance LB (algorithm 7) with $n = 4$ instances and $c = 3$

to meet the SLA (this will be called *first-available-server load-balancing*). This allows for local, application-load-aware load-balancing decisions, without incurring any monitoring. A difference with respect to the mechanism introduced in chapter 7 is that the chain of application instances is fixed – whereas in chapter 7 lists of two pseudo-random candidates are chosen.

Since queries are served by the first available candidate, last instances tend to be less loaded. If too few queries reach the last instance, this indicates that the chain is over-dimensioned; conversely, too many queries reaching the last instance indicates under-dimensioning. Therefore, autoscaling is achieved by the last instance in the chain, rather than by a central entity. By inspecting the quantity of received queries, that last instance will indicate to the load-balancer that another instance must be added to the chain, or that the last one can be removed. This allows for distributed, locally-driven autoscaling decisions, without centralized monitoring.

9.2.1 First-available-instance Load-Balancing

This chapter introduces a load-balancing policy using the framework described in chapter 7, that allows for monitorless up- or down-scaling of a set of n replicated application instances, while ensuring that an SLA is met. A replicated application whose mean service time is $1/\mu$ is considered, for which the expected response time (as seen by clients) T must be lower than an agreed-upon SLA of S (in units of mean service time), *i.e.,* $\mathbf{E}[T] \leq S \times \frac{1}{\mu}$. Then, a simple load-balancing policy consists of ordering the n instances into a chain $(s_1, \ldots, s_n)$, setting a threshold $c = \lfloor S \rfloor$, and making sure that each never handles more than c clients at a given time (see figure 9.1). Formally, each query is assigned $(s_1, \ldots, s_n)$ as a candidate SR list, and each application instance in the list either accepts the query if it currently serves $< c$ clients – otherwise, it forwards the query to the next one in the list (see algorithm 7). To ensure that all queries are served, the last

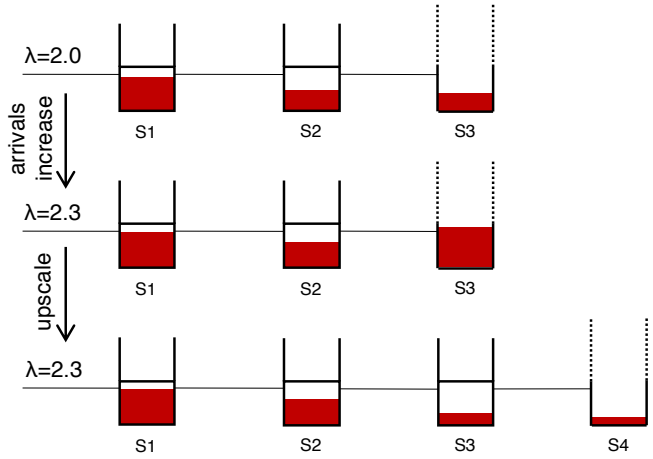


Figure 9.2 – Autoscaling when $c = 3$. The level of red in each instance shows the average number of clients as computed in section 9.4. When the request rate increases from $\lambda = 2.0$ to $\lambda = 2.3$, the third instance observes that is has become highly occupied and thus requests upscaling.

instance s_n must always accept connections. Thus, each of the first $(n-1)$ instances never serve more than c clients, ensuring an expected response time lower than c/μ for queries served by those (assuming either a FIFO or Processor Sharing policy). The last instance accepts the other queries, and, provided that the chain is well-dimensioned, will also serve them within a sufficiently small amount of time – how to dimension the chain is discussed in section 9.2.2.

9.2.2 Autoscaling

A key goal of this load-balancing policy is to make it simple to automatically dimension the size of the chain, while maintaining the SLA: except when the last instance serves too many queries, the SLA will be satisfied by the chain. This can be exploited to perform autoscaling: when the last instance serves too many (or too few) queries, which can be detected locally, this is a signal that the chain must be scaled up (or down) – allowing for monitorless autoscaling, as illustrated in figure 9.2.

As formalized in algorithm 11, the virtual router hosting the last instance in the chain keeps statistics about the size of its queue upon receipt of queries. The fraction of SLA violations (more than c clients in the last instance) is maintained over a fixed time window, and when it goes above a preconfigured threshold ε_{up}, the virtual router hosting the instance requests upscaling of the chain. The virtual router hosting the last instance also maintains the fraction of times it was empty upon query reception, and periodically checks whether this fraction is below a predetermined threshold ε_{down}, in which case downscaling is requested.

9.3 First-Available-Instance LB with 2 Instances

The *first-available-instance* load-balancing scheme introduced in section 9.2.1 is analytically studied in this chapter. For ease of presentation and modeling, the case of $n = 2$ instances (for which the RRR technique applies) is analyzed first. The general case of $n \geq 3$ instances (for which the RRR technique needs to be generalized) will be studied in section 9.4.

9.3.1 Markov Model

Arrivals are assumed to follow to a Poisson process of intensity $\lambda > 0$. Each server has a processing capacity of $\mu > 0$, with exponentially-distributed service times (*i.e.,* the probability of a service lasting less than t is $1 - e^{-\mu t}$). To ease notation, let $\rho = \lambda/\mu$ be the normalized request rate. As described in section 9.2.1, queries are directed to the first instance, which will serve them if the number of pending connections is $< c$, or otherwise pass them to the second instance, which will serve them in any case.

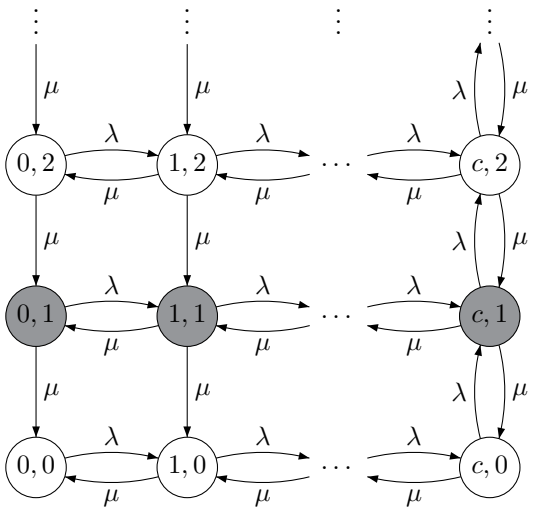


Figure 9.3 – Markov chain for *first-availabe-instance* LB with $n = 2$ instances. State (i, j) means i clients in the first instance and j in the second one. The border row $\{0, \ldots, c\} \times \{1\}$ is grayed.

Notation	Description
$p^D_{(i,j)\to(k,j-1)}$	Probability that first state visited on row $j-1$ when starting from (i, j) is $(k, j-1)$
$p^D_{i\to k}$	Probability that first state visited on row 0 when starting from $(i, 1)$ is $(k, 0)$
$R^D_{(i,j)\to(\star,j-1)}$	Mean reward of $z^{N_2(t)}$ when going from (i, j) down to row $j-1$
R^D_i	Mean reward of $z^{N_2(t)}$ when going from $(i, 1)$ down to row 0
R^L_i	Mean reward of $z^{N_2(t)}$ when going from $(i, 0)$ to $(0, 0)$
R^C	Mean reward of $z^{N_2(t)}$ over a cycle from $(0, 0)$ to $(0, 0)$

Table 9.1 – Notation used when applying the RRR technique

Therefore, the whole system can be modeled as a Markov chain, with state space $\mathcal{S} = \{0, \ldots, c\} \times \mathbb{N}$. In state (i, j), i clients are in the first server's queue and j in the second server's queue. Transitions that increase j can only happen when $i = c$, whereas transitions that increase i can happen from any state with $i < c$. Figure 9.3 shows a graphical representation of this Markov chain.

9.3.2 Applying RRR to the Markov Model

To obtain the client response time, the expected number of clients will be derived [191]. As the behavior of the first server is described by an M/M/1/c law, the probability distribution of N_1, the number of clients served by the first instance, can be derived as:

$$\mathbf{P}[N_1 = k] = \frac{\rho^k}{\sum_{i=0}^{c} \rho^i} = \frac{\rho^k(1-\rho)}{1-\rho^{c+1}},$$

and the expected number of clients in the first instance is:

$$\mathbf{E}[N_1] = \frac{\sum_{k=0}^{c} k\rho^k}{\sum_{k=0}^{c} \rho^k} = \frac{\rho(c\rho^{c+1} - (c+1)\rho^c + 1)}{(\rho-1)(\rho^{c+1}-1)}.$$

To obtain the probability distribution of N_2, the number of clients served by the second instance, the RRR technique is applied with *metric of interest* $M(t) = z^{N_2(t)}$ (where $z \in \mathbb{C}$), as in equation (9.1). This allows deriving the probability generating function of N_2, $f(z) = \mathbf{E}[z^{N_2}]$. The

home state is chosen as $(0,0)$. To compute the mean reward of $M(t)$ over a cycle from the home state to itself, the reward is decomposed into two parts: (i) the mean reward when going one level down in the Markov chain (*i.e.,* until s_2 has one less client), and (ii) the mean reward when going from a state $(i,0)$ to the home state (*i.e.,* until all clients have left). Table 9.1 summarizes the notation used in this chapter.

Reward when going one level down

For $j \geq 1$, let $R^D_{(i,j)\to(\star,j-1)}$ be the mean reward earned between entering state (i,j) and reaching row $j-1$ (*i.e.,* reaching any state $(k,j-1)$). Due to the repeating structure of the chain, that reward only needs to be computed for the *border row* $\{0,\dots,c\}\times\{1\}$ (identified in gray in figure 9.3). This is formalized by the following lemma:

Lemma 9.1. *For $i \in \{0,\dots,c\}$ and $j \geq 1$, $R^D_{(i,j+1)\to(\star,j)} = zR^D_{(i,j)\to(\star,j-1)}$. Therefore, only $R^D_{(i,1)\to(\star,0)}$ needs to be computed, which will be denoted by R^D_i.*

Proof. Since the restriction of the chain to $\{0,\dots,c\}\times\{j,j+1,\dots\}$ (including transitions to the $(j-1)^{\text{th}}$ row) is isomorphic to its restriction to $\{0,\dots,c\}\times\{j+1,j+2,\dots\}$ (including transitions to the j^{th} row), and since there is exactly one more client in the second instance in the latter restriction, the mean reward of $z^{N_2(t)}$ earned when going from $(i,j+1)$ to row j is the same as the mean reward of $z^{N_2(t)+1}$ earned when going from (i,j) to row $j-1$. Hence, $R^D_{(i,j+1)\to(\star,j)} = \mathbf{E}[\int_{(i,j+1)\to(\star,j)} z^{N_2(t)}\mathrm{d}t] = \mathbf{E}[\int_{(i,j)\to(\star,j-1)} z^{N_2(t)+1}\mathrm{d}t] = zR^D_{(i,j)\to(\star,j-1)}$. □

Let $p^D_{(i,j)\to(k,j-1)}$ be the probability that, starting from state (i,j), the first state visited when reaching row $j-1$ is $(k,j-1)$. Since the chain has a recursive structure along the vertical dimension, the following lemma holds:

Lemma 9.2. *For $j \geq 1$ and $k \in \{0,\dots,c\}$, $p^D_{(i,j)\to(k,j-1)}$ is independent from j, and will therefore be denoted by $p^D_{i\to k}$.*

Proof. The proof is similar to that of lemma 9.1. □

It is therefore possible to formulate a system of equations[1] for the $p^D_{i\to k}$, for $k \in \{0,\dots,c\}$:

$$p^D_{0\to k} = \frac{\mu}{\lambda+\mu}\delta_{0k} + \frac{\lambda}{\lambda+\mu}p^D_{1\to k} \tag{9.2}$$

$$p^D_{i\to k} = \frac{\mu}{\lambda+2\mu}\delta_{ik} + \frac{\mu}{\lambda+2\mu}p^D_{i-1\to k} + \frac{\lambda}{\lambda+2\mu}p^D_{i+1\to k} \qquad \forall 1 \leq i \leq c-1 \tag{9.3}$$

$$p^D_{c\to k} = \frac{\mu}{\lambda+2\mu}\delta_{ck} + \frac{\mu}{\lambda+2\mu}p^D_{c-1\to k} + \frac{\lambda}{\lambda+2\mu}\sum_{l=0}^{c} p^D_{c\to l}\, p^D_{l\to k} \tag{9.4}$$

In these equations, the leftmost terms denote the probability of reaching $(k,0)$ from $(i,1)$ directly, while the other terms express reaching $(k,0)$ from $(i,1)$ by transitioning to an adjacent state first. The rightmost term in equation (9.4) comes from the possibility of reaching $(k,0)$ from $(i,1)$ by going to row 2 first, and uses lemma 9.2 to compute the transition from row 2 back to row 0.

Having computed the $p^D_{i\to k}$, it is possible to state a system of equations for the R^D_i:

$$R^D_0 = \frac{z}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu}R^D_1 \tag{9.5}$$

$$R^D_i = \frac{z}{\lambda+2\mu} + \frac{\mu}{\lambda+2\mu}R^D_{i-1} + \frac{\lambda}{\lambda+2\mu}R^D_{i+1} \qquad \forall 1 \leq i \leq c-1 \tag{9.6}$$

$$R^D_c = \frac{z}{\lambda+2\mu} + \frac{\mu}{\lambda+2\mu}R^D_{c-1} + \frac{\lambda}{\lambda+2\mu}\left(zR^D_c + \sum_{k=0}^{c} p^D_{c\to k}\, R^D_k\right) \tag{9.7}$$

[1] δ_{nm} denotes the Kronecker symbol: $\delta_{nm}=1$ if $n=m$ and 0 otherwise.

The first terms in these equations denote the mean reward of z^{N_2} earned when in state $(i,1)$, that is, z times the expected time spent in $(i,1)$, since in these states $N_2 = 1$ and $z^{N_2} = z$. The other terms come from the reward earned if going to an adjacent state instead of directly going to row 0. The bracketed term in equation (9.7) uses lemma 9.1 to express the reward when going from $(c,2)$ to row 1, and then lemma 9.2 for going from a non-deterministic state in row 1 to row 0.

Reward when going from row 0 to the home state

Having computed R_i^D, it is possible to compute R_i^L, the mean reward earned when going from a state $(i,0)$ with $i \geq 1$ to the home state $(0,0)$ – with $R_0^L = 0$, since reaching $(0,0)$ means the end of the cycle. The system of equations for the R_i^L is:

$$R_0^L = 0 \tag{9.8}$$

$$R_i^L = \frac{1}{\lambda+\mu} + \frac{\mu}{\lambda+\mu} R_{i-1}^L + \frac{\lambda}{\lambda+\mu} R_{i+1}^L \qquad \forall 1 \leq i \leq c-1 \tag{9.9}$$

$$R_c^L = \frac{1}{\lambda+\mu} + \frac{\mu}{\lambda+\mu} R_{c-1}^L + \frac{\lambda}{\lambda+\mu} \left(R_c^D + \sum_{j=0}^{c} p_{c\to j}^D \; R_j^L \right) \tag{9.10}$$

This system is obtained in a similar fashion to the R_i^D. Again, the bracketed term in equation (9.10) uses lemma 9.1 and expresses the possibility to go from $(c,0)$ to $(c,1)$, back to one of the possible states in row 0, and finally to $(0,0)$.

Mean reward over a cycle

Finally, it is possible to express R_C, the mean reward over a cycle from the home state to itself:

$$R^C = \frac{1}{\lambda} + R_1^L \tag{9.11}$$

This allows deriving the total expected value of z^{N_2}, using equation (9.1):

$$\mathbf{E}[z^{N_2}] = \frac{\mathbf{E}[\int_{(0,0)\to(0,0)} z^{N_2(t)} \mathrm{d}t]}{\mathbf{E}[\int_{(0,0)\to(0,0)} 1 \mathrm{d}t]} = \frac{R^C(z)}{R^C(1)} \tag{9.12}$$

9.3.3 Closed-form Solution for $c = 1$

When the threshold is $c = 1$ (*i.e.,* s_1 accepts queries only when empty), solving equations (9.2)-(9.12) allows deriving the expected value of N_2 in closed form:

$$\mathbf{E}[N_2] = \frac{\rho^2}{(1+\rho)(\sqrt{1+\rho} - \rho)} \tag{9.13}$$

Proof. The system of equations for the $p_{i\to j}^D$ is the following:

$$p_{0\to 0}^D = \frac{\mu}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu} p_{1\to 0}^D \tag{9.14}$$

$$p_{0\to 1}^D = \frac{\lambda}{\lambda+\mu} p_{1\to 1}^D \tag{9.15}$$

$$p_{1\to 0}^D = \frac{\mu}{\lambda+2\mu} p_{0\to 0}^D + \frac{\lambda}{\lambda+2\mu} (p_{1\to 0}^D p_{0\to 0}^D + p_{1\to 1}^D p_{1\to 0}^D) \tag{9.16}$$

$$p_{1\to 1}^D = \frac{\mu}{\lambda+2\mu} + \frac{\mu}{\lambda+2\mu} p_{0\to 1}^D + \frac{\lambda}{\lambda+2\mu} \left((p_{1\to 1}^D)^2 + p_{1\to 0}^D p_{0\to 1}^D \right) \tag{9.17}$$

Noting that $p_{1\to 0}^D = 1 - p_{1\to 1}^D$, $p_{1\to 1}^D$ can easily by computed by substituting this relation and (9.15) in (9.17), then solving a quadratic equation. $p_{1\to 0}^D$ can then be retrieved from $p_{1\to 0}^D = 1 - p_{1\to 1}^D$, and

$p^D_{0\to 0}, p^D_{0\to 1}$ by substituting back those values in (9.15) and (9.14). This yields, after some algebra:

$$p^D_{0\to 0} = \frac{1}{\sqrt{1+\rho}} \tag{9.18}$$

$$p^D_{0\to 1} = 1 - \frac{1}{\sqrt{1+\rho}} \tag{9.19}$$

$$p^D_{1\to 0} = \frac{1}{\rho}\sqrt{1+\rho} - \frac{1}{\rho} \tag{9.20}$$

$$p^D_{1\to 1} = 1 + \frac{1}{\rho} - \frac{1}{\rho}\sqrt{1+\rho} \tag{9.21}$$

The system of equations for the R^D_i is:

$$R^D_0 = \frac{z}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu} R^D_1 \tag{9.22}$$

$$R^D_1 = \frac{z}{\lambda+2\mu} + \frac{\mu}{\lambda+2\mu} R^D_0 + \frac{\lambda}{\lambda+2\mu}(zR^D_1 + p^D_{1\to 0}\, R^D_0 + p^D_{1\to 1}\, R^D_1) \tag{9.23}$$

Substituting R^D_0, $p^D_{1\to 0}$ and $p^D_{1\to 1}$ in (9.23) and rearranging gives, after some algebra:

$$R^D_1 = \frac{z}{\mu}\frac{1+\rho+\sqrt{1+\rho}}{1+\rho+\sqrt{1+\rho}-\rho(1+\rho)z} \tag{9.24}$$

The equation for R^L_1 is:

$$R^L_1 = \frac{1}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu}\left(R^D_1 + p^D_{1\to 1} R^L_1\right) \tag{9.25}$$

Substituting $p^D_{1\to 1}$ and R^D_1, and solving for R^L_1 yields:

$$R^L_1 = \frac{1}{\mu}\frac{1+\rho+(1+\rho z)\sqrt{1+\rho}}{\sqrt{1+\rho}(1+\rho+\sqrt{1+\rho}-z\rho(1+\rho))} \tag{9.26}$$

From this, one can express the total reward over a cycle, using equation (5.9):

$$R^C = \frac{1}{\lambda}\frac{1+2\rho+\rho^2+(1+2\rho-\rho z)\sqrt{1+\rho}}{\sqrt{1+\rho}(1+\rho+\sqrt{1+\rho}-z\rho(1+\rho))} \tag{9.27}$$

Finally, this allows us to formulate the probability generating function of the number of clients in the second server, using equation (9.12):

$$\mathbf{E}[z^{N_2}] = \frac{(1+\sqrt{\rho+1}-\rho^2)(\rho+\sqrt{1+\rho}+z(1-\sqrt{1+\rho}))}{(1+\rho)(1+\rho+\sqrt{1+\rho}-z\rho(1+\rho))} \tag{9.28}$$

It is finally possible to retrieve the expected number of clients in the second server by taking the derivative of the probability generating function at $z = 1$:

$$\mathbf{E}[N_2] = \frac{\partial \mathbf{E}[z^{N_2}]}{\partial z}\Big|_{z=1} = \frac{\rho^2}{(1+\rho)(\sqrt{1+\rho}-\rho)}$$

□

9.3.4 Solving for $c \geq 2$

When the threshold is $c \geq 2$, it is possible to express the general form for $\mathbf{E}[z^{N_2}]$, and compute numerical[2] values for its coefficients for any value of ρ, as described below.

First, the values of $p^D_{i\to k}$ can be computed numerically by solving the system of equations (9.2),(9.3),(9.4). Plugging these in equations (9.5),(9.6),(9.7) allows computing the R^D_i, yielding the following result:

Lemma 9.3. *For all $i \in \{0,\dots,c\}$, $R^D_i = \frac{A_i z + B_i z^2}{C - Dz}$, where A_i, B_i, C, D are real constants, and $B_c = 0$.*

[2] Unfortunately, computing a closed-form as a function of ρ would involve solving polynomials of degree $c+1$ in ρ.

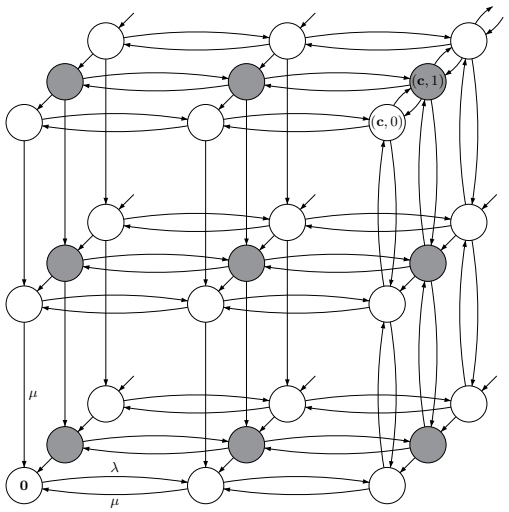


Figure 9.4 – Markov chain for $n = 3$ instances, when $c = 2$. State space is $\{0, 1, 2\}^2 \times \mathbb{N}$. The x, y and z axes represent the first, second and third instances. The border hypercube (a 3×3 square here) $\mathcal{C}_1 = \{0, 1, 2\}^2 \times \{1\}$ is grayed.

Proof. By induction, using (9.5) and (9.6), we can write $R_i^D = \alpha_i R_0^D + \beta_i z$, where α_i, β_i are real constants. In particular, $R_c^D = \alpha_c R_0^D + \beta_c z$. But, (9.7) gives $R_c^D = \frac{Gz + HR_0^D}{I - Jz}$. Combining both forms yields $R_0^D = \frac{Gz - (I-Jz)\beta_c z}{\alpha_c(I-Jz) - H} := \frac{A_0 z + B_0 z^2}{C - Dz}$, and injecting in $R_i^D = \alpha_i R_0^D + \beta_i z$ gives the desired form. For $i = c$, this precisely gives $R_c^D = \frac{\alpha_c Gz - \beta_c z H}{\alpha_c(I-Jz) - H}$, hence $B_c = 0$. □

The values of R_i^D and $p_{i \to k}^D$ can then be inserted into equations (9.9),(9.10) to compute the R_i^L, yielding:

Lemma 9.4. *For all $i \in \{1, \ldots, c\}$, $R_i^L = \frac{E_i + F_i z}{C - Dz}$, where E_i, F_i are real constants, and C, D the same as in lemma 9.3.*

Proof. By induction, we can express the R_i^L as a function of R_1^L, using equation (9.9): $R_i^L = \gamma_i R_1^L + \varepsilon_i$, where γ_i, ε_i are real constants. On the one hand, this gives $R_c^L = \gamma_c R_1^L + \varepsilon_c$. On the other hand, (9.10) gives $R_c^L = K + LR_1^L + MR_c^D$. Combining these two forms gives $R_1^L = N + OR_c^D := \frac{E_1 + F_1 z}{C - Dz}$, hence the desired form for all i by injecting in $R_i^L = \gamma_i R_1^L + \varepsilon_i$. □

This finally allows computing the probability generating function of N_2, as stated in theorem 9.1:

Theorem 9.1. *The probability generating function of N_2 has the form $\mathbf{E}[z^{N_2}] = \frac{E + Fz}{C - Dz}$, where E, F are real constants and C, D are the same as in lemma 9.3. With this notation, the expected number of clients in the second instance is $\mathbf{E}[N_2] = \frac{\partial \mathbf{E}[z^{N_2}]}{\partial z}_{|z=1} = \frac{FC + ED}{(C-D)^2}$.*

Proof. This follows from applying lemma 9.4 to $i = 1$, and using equations (9.11) and (9.12). □

This means that N_2 follows a geometric distribution when conditioned to $N_2 \geq 1$. Combining the expressions for $\mathbf{E}[N_1]$ and $\mathbf{E}[N_2]$ finally yields the expected response time, as per Little's law [191]: $\mathbf{E}[T] = \frac{\mathbf{E}[N_1] + \mathbf{E}[N_2]}{\lambda}$.

9.4 First-Available-Instance LB with $n \geq 3$ Instances

In this section, the model presented in section 9.3 is extended to the case of chains of length $n \geq 3$. With *first-available-instance* load-balancing, a query is accepted by the first instance of the

chain currently serving $< c$ clients, or by the last one otherwise. Hence, the state of the system can be modeled by a vector in $\mathcal{S} = \{0, \dots, c\}^{n-1} \times \mathbb{N}$, where the i-th coordinate represents the number of clients in the i-th server's queue.

In each state, the system can transition *upwards* with rate λ to the state in which the first available instance has *gained* one client, and *downwards* with rate μ to any of those states where one client has *left*. Formally, let $\mathbf{e_i}$ be the vectors of the canonical basis of $\mathbb{R}^n$: $\mathbf{e_i} = (\delta_{i1}, \dots, \delta_{in})$. For each state $\mathbf{x} = (x_1, \dots, x_n)$, let the *upward direction* of $\mathbf{x}$ be $u(\mathbf{x}) = \mathbf{x} + \mathbf{e}_{\min\{i \in \{1,\dots,n\} | x_i < c\}}$, with the convention $\min \emptyset = n$. A set of *downward directions* is also defined as $d(\mathbf{x}) = \bigcup_{i:x_i>0} \{\mathbf{x} - \mathbf{e_i}\}$. With this notation, from each state $\mathbf{x}$ there is a transition to $u(\mathbf{x})$ with rate λ, and a transition to each state in $d(\mathbf{x})$ with rate μ.

Let N_i be the number of clients in the i-th instance. For the first $n-1$ instances, this metric can be evaluated without RRR by solving the balance equations, since the underlying Markov chain has finite state space $\{0, \dots, c\}^{n-1}$. To compute N_n, the number of clients in the last instance, the RRR technique is extended to n-dimensional Markov chains that repeat in one dimension: this new method will be referred to as RRR_n. Let $\mathbf{0} = (0, \dots, 0)$ be the home state. The *metric of interest* is chosen to be $M(t) = z^{N_n(t)}$, the probability generating function of the number of clients in the last instance.

Whereas with RRR_2 a *border row* $\{0, \dots, c\} \times \{1\}$ was identified, from which the reward could be recursively computed, RRR_n uses a *border hypercube* $\mathcal{C}_1 = \{0, \dots, c\}^{n-1} \times \{1\}$. The approach is similar to RRR_2: computing the reward from any state in the border hypercube $\mathcal{C}_1$ down to any state in the base hypercube $\mathcal{C}_0 = \{0, \dots, c\}^{n-1} \times \{0\}$, and from any state of the base hypercube to the home state $\mathbf{0}$. To ease notation, let $\mathcal{C} = \{0, \dots, c\}^{n-1}$ and $\mathbf{c} = (c, \dots, c) \in \mathcal{C}$.

First, it is possible to compute $p^D_{\mathbf{i}\to\mathbf{k}}$, the probability that $\mathbf{y} = (\mathbf{k}, 0) \in \mathcal{C}_0$ is the first state visited in the base hypercube when coming from a state $\mathbf{x} = (\mathbf{i}, 1) \in \mathcal{C}_1$ of the border hypercube, by solving the following system:

$$p^D_{\mathbf{i}\to\mathbf{k}} = \frac{\mu}{\lambda + (1 + |d(\mathbf{i})|)\mu}\delta_{\mathbf{ik}} + \frac{\mu}{\lambda + (1 + |d(\mathbf{i})|)\mu} \sum_{\mathbf{j}\in d(\mathbf{i})} p^D_{\mathbf{j}\to\mathbf{k}} + \frac{\lambda}{\lambda + (1 + |d(\mathbf{i})|)\mu} p^D_{u(\mathbf{i})\to\mathbf{k}}, \forall \mathbf{i} \in \mathcal{C} \setminus \{\mathbf{c}\}, \forall \mathbf{k} \in \mathcal{C} \tag{9.29}$$

$$p^D_{\mathbf{c}\to\mathbf{k}} = \frac{\mu}{\lambda + n\mu}\delta_{\mathbf{ck}} + \frac{\mu}{\lambda + n\mu} \sum_{\mathbf{j}\in d(\mathbf{c})} p^D_{\mathbf{j}\to\mathbf{k}} + \frac{\lambda}{\lambda + n\mu} \sum_{\mathbf{j}\in\mathcal{C}} p^D_{\mathbf{c}\to\mathbf{j}} p^D_{\mathbf{j}\to\mathbf{k}}, \forall \mathbf{k} \in \mathcal{C} \tag{9.30}$$

Then, $R^D_{\mathbf{i}}$, the mean reward earned when going from a state $\mathbf{x} = (\mathbf{i}, 1) \in \mathcal{C}_1$ down to $\mathcal{C}_0$, can be found by solving:

$$R^D_{\mathbf{i}} = \frac{z}{\lambda + (1 + |d(\mathbf{i})|)\mu} + \frac{\mu}{\lambda + (1 + |d(\mathbf{i})|)\mu} \sum_{\mathbf{k}\in d(\mathbf{i})} R^D_{\mathbf{k}} + \frac{\lambda}{\lambda + (1 + |d(\mathbf{i})|)\mu} R^D_{u(\mathbf{i})}, \forall \mathbf{i} \in \mathcal{C} \setminus \{\mathbf{c}\} \tag{9.31}$$

$$R^D_{\mathbf{c}} = \frac{z}{\lambda + n\mu} + \frac{\mu}{\lambda + n\mu} \sum_{\mathbf{k}\in d(\mathbf{c})} R^D_{\mathbf{k}} + \frac{\lambda}{\lambda + n\mu} \left(z R^D_{\mathbf{c}} + \sum_{\mathbf{k}\in\mathcal{C}} p^D_{\mathbf{c}\to\mathbf{k}} R^D_{\mathbf{k}} \right) \tag{9.32}$$

This allows computing $R^L_{\mathbf{i}}$, the mean reward earned when going from a state $\mathbf{x} = (\mathbf{i}, 0) \in \mathcal{C}_0$ to the home state:

$$R^L_{\mathbf{0}} = 0 \tag{9.33}$$

$$R^L_{\mathbf{i}} = \frac{1}{\lambda + |d(\mathbf{i})|\mu} + \frac{\mu}{\lambda + |d(\mathbf{i})|\mu} \sum_{\mathbf{k}\in d(\mathbf{i})} R^L_{\mathbf{k}} + \frac{\lambda}{\lambda + |d(\mathbf{i})|\mu} R^L_{u(\mathbf{i})}, \forall \mathbf{i} \in \mathcal{C} \setminus \{\mathbf{c}, \mathbf{0}\} \tag{9.34}$$

$$R^D_{\mathbf{c}} = \frac{1}{\lambda + (n-1)\mu} + \frac{\mu}{\lambda + (n-1)\mu} \sum_{\mathbf{k}\in d(\mathbf{c})} R^D_{\mathbf{k}} + \frac{\lambda}{\lambda + (n-1)\mu} \left(R^D_{\mathbf{c}} + \sum_{\mathbf{k}\in\mathcal{C}} p^D_{\mathbf{c}\to\mathbf{k}} R^L_{\mathbf{k}} \right) \tag{9.35}$$

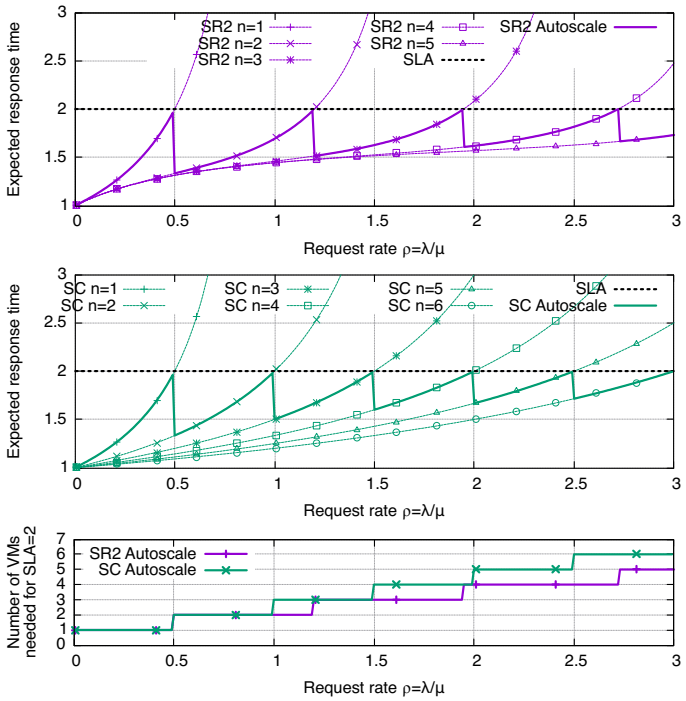


Figure 9.5 – Expected response time $\mathbf{E}[T]$ and number of VMs needed to satisfy SLA, as a function of the request rate. $SLA = 2$. SR with threshold $c = 2$ ($\mathbf{SR_2}$) vs single-choice (**SC**).

Finally, the mean reward of $M(t) = z^{N_n(t)}$ over a cycle, R^C, can be expressed as:

$$R^C = \frac{1}{\lambda} + R^L_{(1,0,\ldots,0)} \tag{9.36}$$

The similar form between equations (9.29)-(9.36) and equations (9.2)-(9.11) allows computing $\mathbf{E}[z^{N_n}]$ similarly as for $n = 2$:

Theorem 9.2. *The probability generating function of N_n has the form $\mathbf{E}[z^{N_n}] = \frac{E_n + F_n z}{C_n - D_n z}$, where C_n, D_n, E_n, F_n are real constants, which depend on c and n.*

9.5 Numerical Results

To evaluate the performance of the load-balancing and autoscaling strategy proposed in this chapter, the expected number of clients in the system is computed as described in section 9.4, allowing to derive the average response time using Little's law [191]: $\mathbf{E}[T] = \mathbf{E}[N]/\lambda$. Note that this computation can only be carried for reasonably small chains, as the non-recursive part of the state space has a size which scales as $\Theta((c+1)^n)$.

As a baseline, random "single-choice" load-balancing is used, for which the response time is:

$$\mathbf{E}[T] = \frac{1}{1 - \rho/n} \frac{1}{\mu} \tag{9.37}$$

Figure 9.5 depicts the average response time for *first-available-instance* load-balancing (denoted by $\mathbf{SR_2}$), with an SLA $S = 2$ and a threshold $c = 2$. In other words, to avoid a response time greater than $S = 2$, all instances but the last serve no more than $c = 2$ clients. (Note that response times and SLAs in this section are described in $1/\mu$ units.) Each of the thin lines shows the response time as a function of the request rate ρ when using n instances, for $n \in \{1, \ldots, 5\}$. The thick line shows the response time when applying autoscaling, assuming a perfect satisfaction of the SLA. As a baseline, figure 9.5 also depicts this response time for *single-choice* load-balancing (**SC**), with the same SLA. As expected, the proposed mechanism allows to use fewer VMs to meet the same SLA as compared to random load-balancing. For instance, the expected response time (in $1/\mu$ units) with $\mathbf{SR_2}$ is 1.83 when $\rho = 1.8$ with $n = 3$ VMs, as compared to 2.5 with $n = 3$

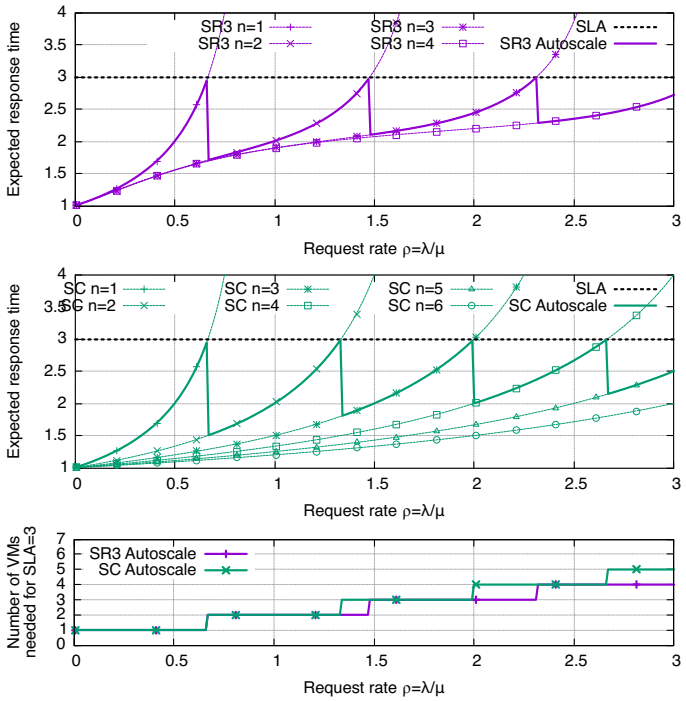


Figure 9.6 – Expected response time $\mathbf{E}[T]$ and number of VMs needed to satisfy SLA, as a function of the request rate. $SLA = 3$. SR with threshold $c = 3$ ($\mathbf{SR_3}$) vs single-choice (**SC**).

VMs or 1.81 with $n = 4$ VMs for single-choice LB. In this example, one more VM is thus needed with single-choice to reach the same service quality. Figure 9.5 (bottom) depicts the number of VMs necessary to meet the SLA, for both policies, confirming that the proposed approach reduces the number of necessary VMs as compared to single-choice LB.

To understand the impact of the SLA on the behavior of the system, figure 9.6 depicts the average response time as a function of the request rate, with a different SLA $S = 3$. The *first-available-instance* policy is used with $c = 3$ as a threshold (SR3), with single-choice load-balancing (**SC**) as baseline. The fairer allocation of queries achieved with SR induces a reduction in the number of VMs needed to achieve a given SLA, as depicted in figure 9.6 (bottom). Due to the less strict SLA, less VMs are required overall as compared the previous example of figure 9.5.

Finally, figure 9.7 depicts the response time and number of VMs needed when the request rate $\rho(t)$ exhibits a diurnal pattern, for $S = 2$ and $c = 2$. With this specific example, never more than 4 VMs are needed to satisfy the SLA with $\mathbf{SR_2}$, whereas **SC** requires up to 6 VMs. The number of VM-hours needed decreases from 83.4 to 66.3, yielding a 21% energy cost reduction.

9.6 Wikipedia Replay

While the results presented in section 9.5 provide analytical insights on the benefits of the proposed architecture, analysis is limited to small chains (due to the increasing size of the state space when n increases). To support the belief that these benefits also hold for larger chains and in real environments, this section presents an implementation of the architecture, and its effects when subject to a real workload.

To that purpose, the load-balancer performing insertion of SR headers and flow steering as described in figure 7.3, and the server agent performing algorithms 7 and 11, are implemented as VPP plugins. As workload, a typical Web environment (Apache, MySQL, PHP, `memcached`) is used to host the MediaWiki software[3], with a dump of the English Wikipedia – similarly as was done in section 7.6.3. This environment is replicated across 24 VMs (each with 2 vCPUs) running on two Intel Xeon E5-2690 machines. The experiment consists of replaying traces of 24 hours

[3] `https://www.mediawiki.org/wiki/Download`

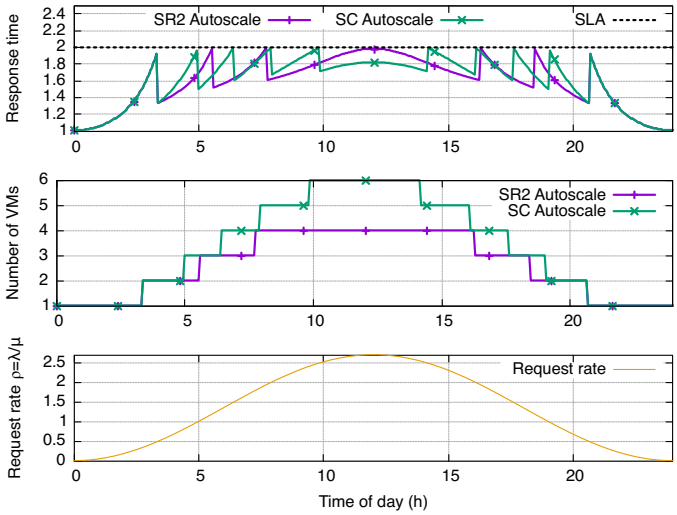


Figure 9.7 – Expected response time $\mathbf{E}[T]$ and number of VMs needed to meet $SLA = 2$, for diurnal request rate pattern $\rho(t) = 2.7(1 + \sin(2\pi(t/24 - 1/4))$. SR with threshold $c = 2$ ($\mathbf{SR_2}$) vs single-choice (**SC**).

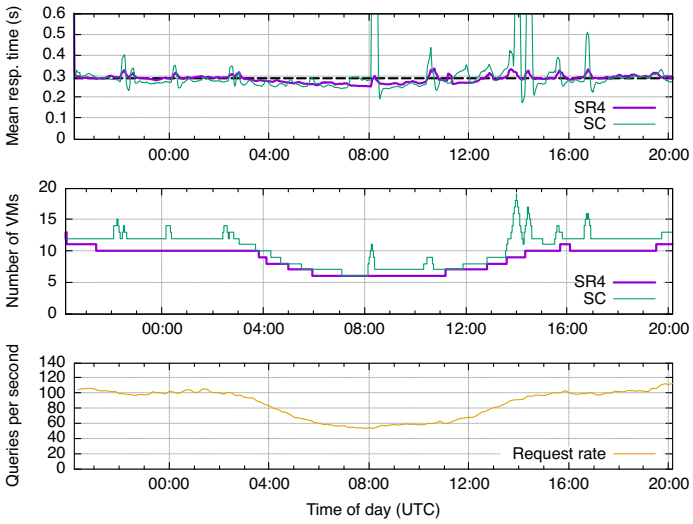


Figure 9.8 – Wikipedia replay: mean response time (moving average with 10 min bins) and number of needed VMs, over 24 hours. SR with $c = 4$, vs **SC** with $SLA = 0.29$ s. For an identical average response time, $\mathbf{SR_4}$ consistently reduces the required number of VMs.

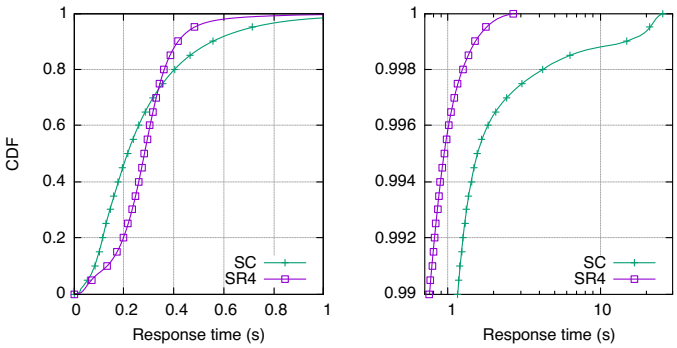


Figure 9.9 – Wikipedia replay: CDF of page load time over 24 hours. SR with $c = 4$, vs **SC** with $SLA = 0.29$ s. For an identical average response time (identical areas over the CDF), $\mathbf{SR_4}$ greatly reduces the tail of the distribution.

Policy	**SR** ($c = 4$)	**SC** ($SLA = 0.29$ s)
Total energy cost (VM-hours)	183	209
Average response time (ms)	289	312
99.9 percentile response time (s)	1.58	17.2

Table 9.2 – SR Autoscaling Wikipedia replay: energy cost and response time

of queries destined to dynamic pages from the English Wikipedia[4] [195], from a traffic generator hosted on a third machine. This trace is interesting not just because it represents real traffic, but also because it exhibits a diurnal pattern, for which autoscaling is desirable.

As a reasonable SLA, $c = 4$ is taken as threshold for algorithm 7 ($\mathbf{SR_4}$). That is, a connection request is forwarded along the chain of application instances, until reaching one with < 4 busy threads in Apache (*i.e.,* twice the number of cores of the VM) – or reaching the last instance. As baseline, Single-Choice randomized load-balancing (**SC**) with a simple feedback autoscaling algorithm is used: response times are collected every two minutes, and if the response time as estimated by equation (9.37) with $n+1$ (respectively $n-1$) VMs is found to be smaller (respectively greater) than a given SLA, upscaling (respectively downscaling) is initiated. For the fairest possible comparison, the 24-hours experiment is first run with $\mathbf{SR_4}$, then the resulting average response time over the day is used as the SLA for the **SC** experiment, ensuring that both policies yield similar average response times.

Results are depicted in table 9.2: while the average response time for all queries is similar with both mechanisms (by design of the experiment), the running cost (in VM-hours) is reduced by 12% when using the proposed mechanism. This is further detailed in figure 9.8, which depicts the request rate as a function of the time of day, alongside the number of running VMs and the instantaneous response time for both policies. The number of VMs automatically follows the request rate, as expected, and with $\mathbf{SR_4}$ the number of running VMs is constantly less than or equal to its baseline counterpart. The gain is greater at peak hours, where SR uses around 10 VMs, *vs* 12 or more for the baseline. Finally, figure 9.9 depicts the distribution of response times over all the queries. In addition to be able to serve queries with the same average response time with fewer VMs, the proposed architecture reduces the tail of the response time distribution (a property highly desirable in data-centers, especially when services are chained and/or parallelized [178]). Notably, the 99^{th} percentile goes from 1.17 s with the baseline to 0.762 s with SR, and the 99.9^{th} percentile from 17.2 s to 1.58 s, an improvement of one order of magnitude.

9.7 Summary of Results

This chapter has introduced a unified load-balancing and autoscaling framework, which allows for local, decentralized decisions. Using Segment Routing, queries are dispatched through a chain of application instances, each of which taking local acceptance decisions, and the last of which taking autoscaling decisions. The proposed architecture yields operational benefits (due to the absence of need for monitoring) as well as service level benefits (as locally-taken decisions provide better load-balancing fairness and autoscaling reactivity). For small chains, benefits in terms of response time and number of instances are demonstrated analytically, by using the Recursive Renewal Reward technique (extended to n-dimensional state spaces). In addition, the proposed architecture has been implemented within a virtual router and evaluated with realistic traffic and workloads, confirming its applicability. Evaluation shows that it is possible to provide the same average response time as with random load-balancing, while decreasing the number of required VMs *and* reducing the tail latency by an order of magnitude, but also removing the need for centralized monitoring to perform autoscaling.

[4]Dynamic pages are found by matching `http://en.wikipedia.org/wiki/index.php/` in the URLs. Due to the capacity of the testbed (which could not sustain replaying every query), every second query was replayed.

Part V

Conclusion

Chapter 10

Conclusion

The growing demand for computing resources has led to the development of complex data center architectures – however, these architectures often provide sub-optimal resource utilization and/or high management overhead, because they fail to provide a tight integration between the applications and the network. This thesis has shown that encoding rich behavior directly at the network layer, and allowing it to transparently interact with applications, is a way to greatly improve scalability of data center architectures. Adopting such an approach has been shown to allow for providing (i) transparent task mobility, (ii) low-footprint reliable multicast, and (iii) in-band load-balancing and autoscaling — therefore allowing for greater scalability and resource usage. This is achieved by architecting network protocols on top of recently-introduced network frameworks (Segment Routing and Bit-Indexed Explicit Replication) which provide source-routing capabilities. Benefits of this approach have been demonstrated by a combination of theoretical analysis, network simulations, and evaluation of real implementations on large-scale testbeds.

In part II, the possibility of performing seamless task mobility in data centers has been investigated. Chapter 3 introduced a framework that enables zero-loss migration of Virtual Machines, using Segment Routing. This is achieved by opening a logical path comprising the source and destination host machines involved in a VM migration, and using SR with special functions to browse this path. This sets the ground for an architecture wherein task can be freely moved according to their communication patterns, rather than placed on one machine once and for all. In chapter 4, this is used as a baseline assumption, to introduce an multi-objective optimization framework aiming at (i) placing the greatest number of tasks, while (ii) migrating already running ones according to their pairwise communication patterns, so as to (iii) maximize overall achievable throughput. A resolution algorithm using the ε-constraint method is used as a baseline, and a heuristic algorithm is proposed, which allows reducing the solving time while providing solutions close to the optimal ones.

Then, part III has investigated how BIER and SR could be used to provide efficient reliable multicast in data centers. Chapter 5 introduced a baseline BIER-based reliable multicast protocol: using Negative Acknowledgements, destinations having missed a copy of a multicast data packet are gathered into a set, and retransmissions are sent to those destinations only, by using a suitable BIER packet. This allows for a lightweight usage of the network resources, as shown by simulations and proven by theoretical analysis. This framework has been further extended in chapter 6, which further diminishes the pressure on costly links, by allowing retransmissions to be sent by peers close to the failing destinations, rather than the source. This is achieved by way of using Segment Routing to send NACKs through a path of local "potential retransmitting peers", each of them deciding on whether they can retransmit a copy of the missed packet, or need to send the NACK to the next candidate.

Finally, part IV has explored how Segment Routing could be used to provide efficient load-balancing services. Chapter 7 set the basis for the part, by introducing 6LB, a load-balancing framework that allows queries to be sent (with SR) through a chain of candidate servers, each of those taking local decisions as to whether to accept the query. This enables load-balancing decisions to be taken by the application themselves, rather than by a central entity which would have to monitor the state of all application instances. This provides non-negligible benefits in terms of client quality of experience (or conversely, in terms of power needed to satisfy a predetermined

quality of experience), as shown by large-scale evaluation on real traces as well as theoretical analysis. Feasibility of a hardware implementation of 6LB has then been explored in chapter 8. Through the use of covert channels, the requirement of per-flow-state inherent to 6LB can be removed, allowing for a stateless implementation. This is validated by the implementation of a prototype for the NetFPGA-SUME platform, demonstrating that it is possible to perform the load-balancing dispatching function with high throughput and negligible latency. Finally, the possibility of expanding the data-plane of 6LB to provide autoscaling has been explored in chapter 9. By sorting application instances into a fixed chain, and sending queries with SR through that chain, the last instance will be naturally under- or over-loaded if the chain itself is under- or over-dimensioned. By observing its local state and reacting accordingly, the last instance can therefore trigger up- or down-scaling, which allows for monitor-less, decentralized auto-scaling. The validity of this approach is confirmed both by mathematical analysis and large-scale evaluation on real traces.

In sum, the work carried in this thesis demonstrates that it is possible to provide generic data-center primitives where decisions are taken *directly in the data-plane*, rather than by a central entity. This therefore goes in complement of traditional Software Defined Networking architectures, wherein the data-plane would be constantly updated to reflect the last decisions taken by the controller, by going one level farther in granularity and realtime-ness. This has been shown to provide better resource usage and lower network usage, while requiring less monitoring and centralization.

To conclude, such an approach provide greater scalability in data center premises, both (i) by unifying the network layer and decreasing the need for out-of-band monitoring, and (ii) by providing better resource utilization as a result of this ability to take local decisions. Such architectures are therefore desirable, for reasons concerning both operational complexity and reduction of energy consumption.

Appendix A

Résumé en français

Cette thèse étudie l'utilisation de protocoles de couche réseau pour fournir, au sein de réseaux de centres de données, des primitives de mobilité des tâches, de distribution fiable de contenu, et d'équilibrage de charge. Elle comprend 5 parties et 10 chapitres, structurés comme suit.

La partie I est une mise en contexte introductive. Le chapitre 1 parcourt l'état de l'art sur les architectures de centres de données (*data centers*) et sur les protocoles réseaux associés. Il introduit ensuite deux architectures réseau intéressantes, IPv6 Segment Routing (SRv6 [7]) et Bit-Indexed Explicit Replication (BIER [8]), qui utilisent le paradigme de routage à la source afin d'ajouter des fonctions à la couche réseau. Le chapitre 2 résume ensuite comment ce concept a été appliqué le long de cette thèse, afin de fournir des primitives de mobilité des tâches, de distribution fiable de contenu, et d'équilibrage de charge.

La partie II étudie la **mobilité des tâches** dans les centres de données. Le chapitre 3 (publié dans [79]) introduit l'utilisation de SRv6 afin de permettre la **migration de machines virtuelles sans perte de paquets**. Les protocoles réseaux traditionnellement utilisés pour la migration de machines virtuelles (*virtual machines*, VMs) reposent sur la signalisation, par une VM venant de terminer une migration, de sa nouvelle localisation à un répertoire centralisé. Cela crée une période de temps transitoire durant laquelle les paquets destinés à de telles VMs sont perdus. Pour résoudre ce problème, le chapitre 3 propose de pré-allouer de manière conservative, grâce à SRv6, un chemin de migration comprenant les machines hôtes de laquelle et vers laquelle la migration est initiée. Ainsi, les paquets atteignent toujours la machine correcte, quelle que soit l'étape au sein du processus de migration. Ce chemin est alloué par le plan de contrôle avant que la migration en elle-même ne soit lancée, et jusqu'après son accomplissement. Ceci est accompli par l'introduction de deux nouvelles fonctions SRv6 : la première vérifie (au sein de la machine hôte source) si le lien vers la VM est toujours établi, et selon le cas, transfère les paquets ou bien vers la VM, ou bien vers le second segment ; la seconde vérifie (au sein de la machine hôte destination) si la VM a repris complètement son exécution, et en fonction, met en tampon localement les paquets, ou bien les transfère vers la VM. Ce mécanisme a été implémenté au sein d'un routeur virtuel (VPP [31]), et l'évaluation montre qu'il est en effet possible de migrer des VMs sans perte de paquets, ce qui permet de réduire la latence et la durée de service des flux servis par l'application hébergée sur la VM.

Une fois la possibilité de migrer des tâches sans perte introduite, le chapitre 4 (publié dans [80]) étudie comment il est possible de fournir de la **migration de tâches utilisant les caractéristiques des flux**, c'est-à-dire de migrer des tâches communicant les unes avec les autres de façon à les rapprocher dans la topologie, et ce pour optimiser le trafic réseau. Les architectures de gestion des centres de données introduites traditionnellement dans la littérature scientifique considèrent ou bien les demandes réseau tâche-à-tâche, ou bien le coût des migrations de tâches, mais pas les deux. Le chapitre 4 introduit un programme d'optimisation multi-objectif visant à maximiser le débit inter-tâche total, tout en minimisation le coût induit par la migration des tâches, et en maximisant le nombre de tâches nouvellement allouées. Un programme non-linéaire en variables mixtes entières (*Mixed Integer Non-Linear Programming*, MILNP) est introduit afin de capturer les contraintes et objectives permettant de modéliser ce problème, et ce programme est ensuite linéarisé en une formulation comme programme linéaire en variables mixtes entières (*Mixed Integer Linear Programming*, MILP). La méthode de l'ε-contrainte est utilisée pour calculer l'ensemble de solutions Pareto-optimales, et sert de référence. À l'aide de simulations, il est démontré que

la méthode proposée augmente efficacement le débit total réalisable, et ceci grâce à la migration de tâches communicant ensemble vers des nœuds plus proches dans la topologie. Du fait du coût élevé en calcul de cette approche, une méthode heuristique est proposée pour calculer de manière incrémentale des solutions au coût de migration croissant, ce qui permet de retourner une approximation de la frontière de Pareto. Des simulations montrent que l'heuristique proposée permet de réduire le temps de calcul des solutions d'un à deux ordres de grandeurs, tout en fournissant des solutions proches de l'optimum.

La partie III étudie la **diffusion fiable de contenus** dans les centres de données. Le chapitre 5 (publié dans [81]) introduit l'utilisation de BIER pour fournir de la *diffusion multipoint fiable et efficace* ("*BIER fiable*"). Les protocoles de diffusion multipoint fiable utilisent traditionnellement un schéma reposant sur des acquittements négatifs (*Negative-ACKnowledgements*, NACKs), par lequel les destinations perdant des paquets le signalent à la source. La source utilise ensuite ces signaux pour produire ou bien des retransmissions point-à-point vers chacune des destinations ayant perdu le paquet, ou bien des retransmissions multipoint vers l'ensemble du groupe. Le chapitre 5 propose un protocole reposant sur des NACKs qui vise à minimiser l'impact du trafic de retransmission, en utilisant BIER. Au lieu d'utiliser des chaînes de bits (*bitstrings*) différentes pour chaque flux, des chaînes de bits différentes sont utilisées *pour chaque paquet* : une retransmission pour un paquet perdu est transmise uniquement à l'ensemble des destinations qui n'ont pas reçu ce paquet. Pour accomplir cela, la source rassemble les NACKs envoyés par les différentes destinations pendant un court intervalle de temps, et ce afin de construire une chaîne de bits appropriée pour la retransmission. Ce protocole a été implémenté dans un simulateur de trafic réseau (`ns3` [147]), ce qui a permis de conduire une campagne de simulations dans différentes topologies, avec différents scénarios de perte de paquets. Les simulations montrent que, surtout dans le cas où un sous-ensemble du centre de données est caractérisé par des pertes localisées, le trafic total émis par le protocole est réduit (i) comparé aux retransmissions multipoint, car l'arbre tout entier n'a pas à être inondé en cas de perte, et (ii) comparé aux retransmissions point-à-point, car le trafic n'est pas dupliqué sur les liens limitants. Un modèle mathématique vient ensuite compléter l'analyse, permettant ainsi de quantifier l'empreinte du traffic de retransmission pour les trois mécanismes (retransmissions point-à-point, multipoint, et BIER), dans des topologies arborescentes arbitraires. Une approximation au premier ordre en $\alpha \to 0$ (où α est le taux de perte de paquets sur les liens) est calculée dans le théorème 5.1. Cette approximation montre que le trafic dû aux retransmissions se comporte en $L \log L$ avec les retransmissions BIER, comparé à $L \log^2 L$ pour les retransmissions point-à-point, et L^2 pour les retransmissions multipoint (où L est le nombre de liens de l'arbre) : ceci confirme théoriquement les bénéfices précédemment mentionnés.

Le chapitre 6 (soumis en tant que [82]) étend l'utilisation du *BIER fiable* pour fournir des **retransmissions multipoint assistées par des pairs**. Une extension au plan de données de BIER est proposée, afin de permettre aux destinations d'apprendre quels pairs (*i.e.,* des destinations ayant souscrit au même flux multipoint) sont proches topologiquement. Ceci est accompli grâce à une *chaîne de bits des pairs* (*peerstring*) incluse dans chaque paquet, qui est mise à jour par chaque nœud afin qu'elle contienne invariablement l'ensemble des destinations atteintes par ce nœud et pour ce paquet. Ensuite, lorsqu'un paquet est perdu, SRv6 est utilisé par les destinations n'ayant pas reçu le paquet afin de diriger les NACKs à travers un chemin comprenant un ou plusieurs pairs et la source, jusqu'à trouver un pair (ou en dernier recours, la source) capable de transmettre une retransmission du paquet perdu. Différentes politiques de sélection des pairs sont proposées et évaluées mathématiquement. En premier lieu, deux politiques statiques simples sont analysées : la sélection aléatoire d'un pair dans le sous-arbre d'une destination, et la sélection d'un pair pré-désigné dans ce même sous-arbre. L'analyse mathématique révèle que la seconde politique génère moins de trafic de retransmission (car les retransmissions sont envoyées uniquement par un pair), mais a moins de chance de réussir (car si le pair pré-désigné n'a pas reçu le paquet non plus, aucun pair ne recevra de retransmission). En second lieu, une politique dynamique est introduite, selon laquelle chaque pair essaie d'apprendre dynamiquement de quel autre pair il a le plus de chance d'obtenir une retransmission. L'architecture proposée est évaluée grâce à des simulations réseaux paquet par paquet, dans différentes topologies et avec différentes politiques de sélection des pairs. L'évaluation montre qu'en utilisant cette architecture, il est possible de réduire la charge sur les liens centraux (comparé au chapitre 5) en augmentant la localité des retransmissions, ce qui permet ainsi de réduire l'empreinte totale du trafic réseau.

La partie IV étudie l'**équilibrage de charge** dans les centres de données. Le chapitre 7 (publié dans [83, 84]) introduit l'utilisation de **SRv6 pour l'équilibrage de charge utilisant les**

charges instantanées des applications. Traditionnellement, les équilibreurs de charges des centres de données sont répliqués, à des fins de fiabilité et de passage à l'échelle. Lorsque c'est le cas, le hachage cohérent (*consistent hashing*) est utilisé, afin de s'assurer qu'un flux est toujours assigné au même serveur le long de son existence, et ce quel que soit l'équilibreur de charge par lequel il est traité, et même en cas de reconfiguration de l'ensemble des instances d'application. Cependant, une telle approche fait que les requêtes sont assignées aux instances d'application sans prendre en compte leur charge instantanée. Pour remédier à cela, le chapitre 7 introduit l'utilisation de SRv6 pour diriger le premier paquet (TCP SYN) d'une requête au travers d'une chaîne de *deux applications candidates aléatoires*, chacune d'entre elles décidant successivement si elle peut ou non accepter la requête, selon son état local. Ceci permet d'avoir un mécanisme d'équilibrage de charge qui prend en compte les états courants des applications, directement au sein de la couche réseau, et sans recourir à une observation centralisée de l'état des applications. Pour éviter un trafic triangulaire non nécessaire, les paquets suivants sont envoyés directement à l'instance ayant accepté la connection. Ceci est accompli grâce à un protocole de signalisation intra-bande, en utilisant des fonctions SRv6 dédiées entre les équilibreurs de charges et les serveurs afin d'installer et de retirer l'état nécessaire pour chaque connexion. La fiabilité du mécanisme au regard des changements de l'ensemble des équilibreurs de charge et/ou des applications est assurée grâce à un hachage cohérent qui assigne les flux à des listes (et non pas des singletons) d'instances d'applications. Un modèle mathématique du temps de réponse observé par les clients est ensuite introduit, pour des arrivées Poisson et des services suivant une loi exponentielle. Grâce à ce modèle, il est possible de quantifier les bénéfices de l'approche proposée en termes d'espérance du temps de réponse (ainsi que de la queue de la distribution correspondante), d'équité de charge des serveurs, et de réduction d'énergie. L'analyse démontre que, sous l'hypothèse raisonnable que le délai réseau unidirectionnel entre deux machines est inférieur au temps de service moyen, les performances en temps de réponse sont toujours améliorées, comparé à de l'équilibrage de charge aléatoire (théorème 7.1). L'architecture proposée a été implémentée comme module pour un routeur virtuel (VPP), qui extrait de manière transparente des informations d'un serveur HTTP standard (Apache) grâce à une mémoire partagée, et utilise ces informations pour prendre des décisions d'acceptation de connexion. Une évaluation conduite sur une plateforme d'essai comprenant 48 instances, et utilisant à la fois du trafic synthétique et des traces réelles, confirme ces bénéfices. L'évaluation montre notamment qu'il est possible de fournir la même qualité de service moyenne aux clients (*i.e.,* le même temps de réponse moyen), tout en réduisant le nombre de machines virtuelles nécessaires de 17% comparé à l'équilibrage de charge aléatoire.

Le chapitre 8 (publié dans [85]) explore ensuite la faisabilité d'implémenter, dans des périphériques matériels, une architecture d'**équilibrage de charge sans état et utilisant les charges instantanées des applications**. En effet, implémenter des équilibreurs de charge dans des périphériques matériels est désirable pour des raisons d'efficacité et de passage à l'échelle. Cependant, ceci est incompatible avec le maintien d'état par connexion, ce qui est notamment requis par les architectures utilisant les charges instantanées des applications. Pour contourner ce problème, le chapitre 8 explore l'utilisation de canaux cachés (*covert channels*) pour transporter cet état directement dans les en-têtes des paquets. Les connexions sont établies grâce au même mécanisme que dans le chapitre 7 : les paquets d'établissement de connexion sont envoyés à travers une chaîne d'applications candidates, jusqu'à ce que l'une d'entre elles accepte (selon son état instantané) de servir la requête. Cependant, au lieu d'installer de l'état dans l'équilibreur de charge, l'instance qui a accepté la connexion communique au client sa *position* dans la liste SRv6, via un canal caché. Cette valeur est alors automatiquement reflétée par le client jusqu'à atteindre l'équilibreur de charge, qui peut ensuite utiliser cette information pour diriger le paquet vers le serveur ayant initialement accepté la connexion. Pour réaliser un tel canal caché en pratique et permettre son utilisation par des clients standards et non-modifiés, le chapitre 8 propose d'utiliser les bits de poids faible des estampille de temps (*timestamps*) du protocole TCP. De plus, l'équilibreur de charge s'assure de la fiabilité du système en cas de changement de l'ensemble des applications, grâce à un versionnage des tables de hachage cohérent – ainsi, les paquets se voient insérer un en-tête SRv6 comprenant les instances qui étaient précédemment retournées par la fonction de hachage cohérent. Un prototype a été réalisé sur une carte réseau reprogrammable (utilisant le langage P4 [200] et ciblant la plateforme NetFPGA-SUME [201]), prourant la faisabilité d'implémentation de cette architecture. Des simulations paquet par paquet de cette implémentation montrent que la latence due à l'analyse des en-têtes des paquets est négligeable (de l'ordre de 10 μs). Des simulations du mécanisme de hachage cohérent proposé montrent que la fiabilité est améliorée d'un ordre de

grandeur, comparé à des mécanismes n'utilisant pas le versionnage. En résumé, cette architecture permet d'obtenir les mêmes bénéfices que ceux introduits au chapitre 7, tout en bénéficiant d'une implémentation matérielle à faible latence et haut débit, et en améliorant la fiabilité du hachage cohérent.

Le chapitre 9 (soumis en tant que [86]) introduit une architecture d'**auto-équilibrage sans observation centralisée**, utilisant SRv6. Afin de satisfaire les variations journalières en demande de trafic tout en satisfaisant un niveau de service prédéfini, les centres de données utilisent traditionnellement l'auto-équilibrage (*autoscaling*) afin d'adapter en temps réel le nombre d'instances répliquées d'un service donné. Les auto-équilibreurs utilisent des architectures centralisées pour prendre ces décisions, ce qui induit un surcoût d'observation. Dès lors, le chapitre 9 étudie l'utilisation de SRv6 pour fournir un service commun de distribution de charge et d'auto-équilibrage, sans observation centralisée. Les instances de l'application répliquée sont ordonnées le long d'une chaîne fixe, et les nouvelles requêtes sont envoyées le long de cette chaîne, jusqu'à ce que l'une d'entre elles accepte de servir la connexion – et ce en utilisant le plan de données introduit au chapitre 7. De plus, la dernière instance de la chaîne garde trace de l'historique de son propre état de charge afin de déclencher l'ajout ou le retrait d'un serveur en fin de chaîne. En effet, le fait que la dernière instance de la chaîne reçoive trop peu (ou trop) de requêtes est un indicateur que la chaîne est sur-provisionnée (ou sous-provisionnée). Un modèle de Markov de la performance du système est introduit, et permet d'exprimer stochastiquement le nombres de requêtes servies par chaque serveur. Ce modèle est résolu grâce à la technique RRR (*Recursive Renewal Reward* [208]), ce qui permet d'exprimer, pour de petites chaînes, l'espérance du temps de réponse perçu par les clients – et ainsi d'en déduire les gains possibles d'énergie offerts par le système (*i.e.,* la réduction possible du nombre de serveurs requis pour assurer la même qualité de service qu'avec de l'équilibrage de charge aléatoire). L'architecture a été implémentée en tant que module pour VPP, et une évaluation avec des traces de trafic confirme que ces bénéfices peuvent également être observés dans des environnements réels. Ainsi, comparé à l'équilibrage de charge aléatoire, le même temps de réponse moyen peut être offert aux clients, tout en réduisant le nombre de machines virtuelles nécessaires, et en diminuant simultanément la queue de la distribution de temps de service.

Enfin, le manuscrit de thèse se termine par une conclusion donnée en partie V.

List of Figures

This thesis comprises 89 figures, listed below.

List of Tables

This thesis comprises 15 tables, listed below.

List of Algorithms

Bibliography

[1] R. P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34–45, 1974.

[2] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[3] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[4] N. Bitar, S. Gringeri, and T. J. Xia, "Technologies and protocols for data center and cloud networking," *IEEE Communications Magazine*, vol. 51, no. 9, pp. 24–31, 2013.

[5] B. Wang, Z. Qi, R. Ma, H. Guan, and A. V. Vasilakos, "A survey on data center networking for cloud computing," *Computer Networks*, vol. 91, pp. 528–547, 2015.

[6] K. Bilal, S. U. R. Malik, O. Khalid, A. Hameed, E. Alvarez, V. Wijaysekara, R. Irfan, S. Shrestha, D. Dwivedy, M. Ali *et al.*, "A taxonomy and survey on green data center networks," *Future Generation Computer Systems*, vol. 36, pp. 189–208, 2014.

[7] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, "The Segment Routing architecture," in *Proc. IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–6.

[8] I. Wijnands, E. C. Rosen, A. Dolganow, T. Przygienda, and S. Aldrin, "Multicast Using Bit Index Explicit Replication (BIER)," RFC 8279, 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8279.txt

[9] "Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model," ISO/IEC 7498-1, Geneva, Nov. 1994.

[10] W. Goralski, *The illustrated network: how TCP/IP works in a modern network*. Burlington, MA: Morgan Kaufmann, 2017.

[11] "IEEE Standard for Ethernet," IEEE 802.3-2015, Piscataway, NJ, USA, Mar. 2016.

[12] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall, "Augmenting data center networks with multi-gigabit wireless links," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 38–49.

[13] J. Postel, "Internet Protocol," RFC 791, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc791.txt

[14] S. Deering and B. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 8200, Jul. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8200.txt

[15] Y. Rekhter, S. Hares, and T. Li, "A Border Gateway Protocol 4 (BGP-4)," RFC 4271, Jan. 2006. [Online]. Available: https://rfc-editor.org/rfc/rfc4271.txt

[16] P. Lapukhov, A. Premji, and J. Mitchell, "Use of BGP for Routing in Large-Scale Data Centers," RFC 7938, Aug. 2016. [Online]. Available: https://rfc-editor.org/rfc/rfc7938.txt

[17] J. Postel, "Transmission Control Protocol," RFC 793, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc793.txt

[18] H. F. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, Jun. 1999. [Online]. Available: https://rfc-editor.org/rfc/rfc2616.txt

[19] E. Rescorla, "HTTP Over TLS," RFC 2818, May 2000. [Online]. Available: https://rfc-editor.org/rfc/rfc2818.txt

[20] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[21] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.

[22] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang *et al.*, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[23] Y. Huang, B. Gao, W. Sun, Z. H. Wang, and C. J. Guo, "A framework for native multi-tenancy application development and management," in *Proc. 9th IEEE International Conference on e-Commerce Technology and 4th IEEE International Conference on Enterprise Computing, e-Commerce, and e-Services (CEC-EEE)*, Jul. 2007, pp. 551–558.

[24] "IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks," IEEE Std 802.1Q-2014, Piscataway, NJ, USA, Dec 2014.

[25] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks," RFC 7348, Aug. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7348.txt

[26] P. Garg and Y.-S. Wang, "NVGRE: Network Virtualization Using Generic Routing Encapsulation," RFC 7637, Sep. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7637.txt

[27] Y. Rekhter and K. Kompella, "Virtual Private LAN Service (VPLS) Using BGP for Auto-Discovery and Signaling," RFC 4761, Jan. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc4761.txt

[28] J. Drake, W. Henderickx, A. Sajassi, R. Aggarwal, D. N. N. Bitar, A. Isaac, and J. Uttaro, "BGP MPLS-Based Ethernet VPN," RFC 7432, Feb. 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7432.txt

[29] A. Viswanathan, E. C. Rosen, and R. Callon, "Multiprotocol Label Switching Architecture," RFC 3031, Jan. 2001. [Online]. Available: https://rfc-editor.org/rfc/rfc3031.txt

[30] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of Open VSwitch," in *Proc. 12th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2015, pp. 117–130.

[31] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi, "High-speed software data plane via vectorized packet processing," *IEEE Communications Magazine*, vol. 56, no. 12, pp. 97–103, December 2018.

[32] The DPDK Project, "Data Plane Development Kit (DPDK)." [Online]. Available: https://www.dpdk.org

[33] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[34] *Cisco Data Center Infrastructure 2.5 Design Guide.* San Jose, CA: Cisco, Nov. 2011.

[35] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 63–74.

[36] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: a scalable and fault-tolerant network structure for data centers," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 75–86.

[37] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "BCube: a high performance, server-centric network architecture for modular data centers," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 63–74, 2009.

[38] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. 100, no. 10, pp. 892–901, 1985.

[39] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: a scalable and flexible data center network," in *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[40] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Burlington, MA: Morgan Kaufmann, 2004.

[41] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM conference on Internet measurement (IMC)*. ACM, Nov. 2010, pp. 267–280.

[42] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *Proc. 7th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, vol. 10, 2010, pp. 19–19.

[43] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[44] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[45] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 15–26.

[46] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 503–514.

[47] C. Filsfils, S. Previdi, J. Leddy, M. S., and V. Daniel, "IPv6 Segment Routing Header (SRH)," Internet Engineering Task Force, Internet-Draft draft-ietf-6man-segment-routing-header-14, Jun. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-6man-segment-routing-header-14

[48] S. Previdi, C. Filsfils, B. Decraene, S. Litkowski, M. Horneffer, and R. Shakir, "Source Packet Routing in Networking (SPRING) Problem Statement and Requirements," RFC 7855, May 2016. [Online]. Available: https://rfc-editor.org/rfc/rfc7855.txt

[49] A. Bashandy, C. Filsfils, S. Previdi, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing with MPLS data plane," Internet Engineering Task Force, Internet-Draft draft-ietf-spring-segment-routing-mpls-14, Jun. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-spring-segment-routing-mpls-14

[50] C. Filsfils, P. C. Garvia, J. Leddy, V. Daniel, M. S., and Z. Li, "SRv6 Network Programming," Internet Engineering Task Force, Internet-Draft draft-filsfils-spring-srv6-network-programming-05, Jul. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-filsfils-spring-srv6-network-programming-05

[51] B. Hinden and D. S. E. Deering, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, Dec. 1998. [Online]. Available: https://rfc-editor.org/rfc/rfc2460.txt

[52] G. Neville-Neil, P. Savola, and J. Abley, "Deprecation of Type 0 Routing Headers in IPv6," RFC 5095, Dec. 2007. [Online]. Available: https://rfc-editor.org/rfc/rfc5095.txt

[53] P. Biondi and A. Ebalard, "IPv6 routing header security," in *Proc. CanSecWest Security Conference*, Apr. 2007.

[54] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois, "A declarative and expressive approach to control forwarding paths in carrier-grade networks," in *ACM SIGCOMM computer communication review*, vol. 45, no. 4. ACM, 2015, pp. 15–28.

[55] R. Bhatia, F. Hao, M. Kodialam, and T. Lakshman, "Optimized network traffic engineering using segment routing," in *Proc. IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 2015, pp. 657–665.

[56] S. Gay, R. Hartert, and S. Vissicchio, "Expect the unexpected: Sub-second optimization for segment routing," in *Proc. IEEE Conference on Computer Communications (INFOCOM)*, May 2017, pp. 1–9.

[57] A. Cianfrani, M. Listanti, and M. Polverini, "Incremental deployment of segment routing into an ISP network: a traffic engineering perspective," *IEEE/ACM Transactions on Networking*, vol. 25, no. 5, pp. 3146–3160, Oct. 2017.

[58] D. Lebrun, M. Jadin, F. Clad, C. Filsfils, and O. Bonaventure, "Software resolved networks: Rethinking enterprise networks with IPv6 segment routing," in *Proc. ACM Symposium on SDN Research (SOSR)*. ACM, 2018, article no. 6.

[59] D. Lebrun, "Leveraging IPv6 segment routing for service function chaining," in *Proc. CoNEXT Student Workshop*, 2015, pp. 1–15.

[60] A. AbdelSalam, F. Clad, C. Filsfils, S. Salsano, G. Siracusano, and L. Veltri, "Implementation of virtual network function chaining through segment routing in a Linux-based NFV infrastructure," in *Proc. IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2017, pp. 1–5.

[61] A. Abdelsalam, S. Salsano, F. Clad, P. Camarillo, and C. Filsfils, "SERA: Segment routing aware firewall for service function chaining scenarios," in *Proc. IFIP Networking Conference (IFIP Networking)*. IEEE, May 2018, pp. 1–9.

[62] F. Duchêne, D. Lebrun, and O. Bonaventure, "SRv6Pipes: enabling in-network bytestream functions," in *Proc. IFIP Networking Conference (IFIP Networking)*. IEEE, May 2018, pp. 1–9.

[63] F. Aubry, D. Lebrun, S. Vissicchio, M. T. Khong, Y. Deville, and O. Bonaventure, "SCMon: Leveraging segment routing to improve network monitoring," in *Proc. IEEE Conference on Computer Communications (INFOCOM)*, Apr. 2016, pp. 1–9.

[64] F. Aubry, D. Lebrun, Y. Deville, and O. Bonaventure, "Traffic duplication through segmentable disjoint paths," in *Proc. IFIP Networking Conference (IFIP Networking)*. IEEE, 2015, pp. 1–9.

[65] D. Lebrun and O. Bonaventure, "Implementing IPv6 segment routing in the linux kernel," in *Proc. Applied Networking Research Workshop (ANRW)*. ACM, 2017, pp. 35–41.

[66] J. Nicholas, A. Adams, and W. Siadak, "Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised)," RFC 3973, 2005. [Online]. Available: https://rfc-editor.org/rfc/rfc3973.txt

[67] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen, "Deployment issues for the IP multicast service and architecture," *IEEE Network*, vol. 14, no. 1, pp. 78–88, 2000.

[68] I. Wijnands, E. C. Rosen, A. Dolganow, J. Tantsura, S. Aldrin, and I. Meilik, "Encapsulation for Bit Index Explicit Replication (BIER) in MPLS and Non-MPLS Networks," RFC 8296, Jan. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8296.txt

[69] J. Xie, L. Geng, M. McBride, S. Dhanaraj, G. Yan, and Y. Xia, "Encapsulation for BIER in Non-MPLS IPv6 Networks," Internet Engineering Task Force, Internet-Draft draft-xie-bier-ipv6-encapsulation-00, Mar. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-xie-bier-ipv6-encapsulation-00

[70] P. Pfister and I. Wijnands, "An IPv6 based BIER Encapsulation and Encoding," Internet Engineering Task Force, Internet-Draft draft-pfister-bier-over-ipv6-01, Oct. 2016, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-pfister-bier-over-ipv6-01

[71] W. Braun, M. Albert, T. Eckert, and M. Menth, "Performance comparison of resilience mechanisms for stateless multicast using BIER," in *Proc. IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 230–238.

[72] J. Papán, P. Segeč, M. Drozdová, L. Mikuš, M. Moravčík, and J. Hrabovskỳ, "The IPFRR mechanism inspired by BIER algorithm," in *Proc. International Conference on Emerging eLearning Technologies and Applications (ICETA)*. IEEE, 2016, pp. 257–262.

[73] A. Giorgetti, A. Sgambelluri, F. Paolucci, P. Castoldi, and F. Cugini, "First demonstration of SDN-based Bit Index Explicit Replication (BIER) multicasting," in *Proc. European Conference on Networks and Communications (EuCNC)*, Jun. 2017, pp. 1–6.

[74] A. Giorgetti, A. Sgambelluri, F. Paolucci, N. Sambo, P. Castoldi, and F. Cugini, "Bit Index Explicit Replication (BIER) multicasting in transport networks," in *Proc. International Conference on Optical Network Design and Modeling (ONDM)*. IEEE, 2017, pp. 1–5.

[75] Z. Brodard, H. Jiang, T. Chang, T. Watteyne, X. Vilajosana, P. Thubert, and G. Texier, "Rover: Poor (but elegant) man's testbed," in *Proc. 13th ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks*. ACM, 2016, pp. 61–65.

[76] W. Braun, J. Hartmann, and M. Menth, "Scalable and reliable software-defined multicast with BIER and P4," in *Proc. IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 905–906.

[77] D. Saucez, L. Iannone, O. Bonaventure, and D. Farinacci, "Designing a deployable internet: the locator/identifier separation protocol," *Internet Computing, IEEE*, vol. 16, no. 6, pp. 14–21, 2012.

[78] T. Herbert and P. Lapukhov, "Identifier-locator addressing for IPv6," Internet Engineering Task Force, Internet-Draft draft-herbert-intarea-ila-01, Mar. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-herbert-intarea-ila-01

[79] Y. Desmouceaux, M. Townsley, and T. Clausen, "Zero-loss virtual machine migration in IPv6 data-centers with segment routing," in *Proc. 14th International Conference on Network and Service Management (CNSM), 1st Workshop on Segment Routing and Service Function Chaining (SR+SFC)*. IEEE, 2018, pp. 420–425.

[80] Y. Desmouceaux, S. Toubaline, and T. Clausen, "Flow-aware workload migration in data centers," *Journal of Network and Systems Management*, vol. 26, no. 4, pp. 1034–1057, Oct. 2018.

[81] Y. Desmouceaux, T. Clausen, J.-A. Cordero Fuertes, and W. M. Townsley, "Reliable multicast with B.I.E.R." *Journal of Communications and Networks*, vol. 20, no. 2, pp. 182–197, 2018.

[82] Y. Desmouceaux, J.-A. Cordero Fuertes, and T. Clausen, "Reliable BIER with peer caching," Submitted, 2019.

[83] Y. Desmouceaux, P. Pfister, J. Tollet, M. Townsley, and T. Clausen, "SRLB: The power of choices in load balancing with segment routing," in *Proc. IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2011–2016.

[84] ——, "6LB: Scalable and application-aware load balancing with segment routing," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 819–834, Apr. 2018.

[85] B. Pit-Claudel, Y. Desmouceaux, P. Pfister, M. Townsley, and T. Clausen, "Stateless load-aware load balancing in P4," in *Proc. IEEE 26th International Conference on Network Protocols (ICNP), 1st P4 Workshop in Europe (P4WE)*. IEEE, 2018, pp. 418–423.

[86] Y. Desmouceaux, M. Enguehard, and T. Clausen, "Joint monitorless load-balancing and autoscaling for zero-wait-time in data centers," Submitted, 2019.

[87] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proc. 2nd USENIX Symposium on Networked Systems Design & Implementation (NSDI)*. USENIX Association, 2005, pp. 273–286.

[88] R. Bolla, M. Chiappero, R. Rapuzzi, and M. Repetto, "Seamless and transparent migration for TCP sessions," in *Proc. IEEE 25th Annual International Symposium on Personal, Indoor, and Mobile Radio Communication (PIMRC)*. IEEE, 2014, pp. 1469–1473.

[89] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete container state migration," in *Proc. IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2137–2142.

[90] R. W. Ahmad, A. Gani, S. H. A. Hamide, M. Shiraz, A. Yousafzai, and F. Xia, "A survey on virtual machine migration and server consolidation frameworks for cloud data centers," *Journal of Network and Computer Applications*, vol. 52, pp. 11–25, 2015.

[91] C. Ghribi, M. Hadji, and D. Zeghlache, "Energy efficient VM scheduling for cloud data centers: Exact allocation and migration algorithms," in *Proc. 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*. IEEE, 2013, pp. 671–678.

[92] D. Kakadia, N. Kopri, and V. Varma, "Network-aware virtual machine consolidation for large data centers," in *Proc. 3rd International Workshop on Network-Aware Data Management (NDM)*. ACM, 2013, article no. 6.

[93] D. Zeng, L. Gu, and S. Guo, "Cost minimization for big data processing in geo-distributed data centers," in *Cloud Networking for Big Data*. Springer, 2015, pp. 59–78.

[94] Y. Wang, E. Keller, B. Biskeborn, J. Van Der Merwe, and J. Rexford, "Virtual routers on the move: live router migration as a network-management primitive," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 231–242.

[95] D. B. Johnson, J. Arkko, and C. E. Perkins, "Mobility Support in IPv6," RFC 6275, Jul. 2011. [Online]. Available: https://rfc-editor.org/rfc/rfc6275.txt

[96] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, "Live wide-area migration of virtual machines including local persistent state," in *Proc. 3rd international conference on Virtual execution environments*. ACM, 2007, pp. 169–179.

[97] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[98] Q. Li, J. Huai, J. Li, T. Wo, and M. Wen, "HyperMIP: Hypervisor controlled mobile IP for virtual machine live migration across networks," in *Proc. 11th IEEE High Assurance Systems Engineering Symposium (HASE)*. IEEE, 2008, pp. 80–88.

[99] H. Watanabe, T. Ohigashi, T. Kondo, K. Nishimura, and R. Aibara, "A performance improvement method for the global live migration of virtual machine with IP mobility," in *Proc. 5th International Conference on Mobile Computing and Ubiquitous Networking (ICMU)*, vol. 94, 2010, pp. 1–6.

[100] P. Raad, S. Secci, D. C. Phung, A. Cianfrani, P. Gallard, and G. Pujolle, "Achieving sub-second downtimes in large-scale virtual machine migrations with LISP," *IEEE Transactions on Network and Service Management*, vol. 11, no. 2, pp. 133–143, 2014.

[101] U. Kalim, M. K. Gardner, E. J. Brown, and W.-c. Feng, "Seamless migration of virtual machines across networks," in *Proc. 22nd International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2013, pp. 1–7.

[102] D. M. F. Mattos and O. C. M. B. Duarte, "XenFlow: Seamless migration primitive and quality of service for virtual networks," in *Proc. IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2014, pp. 2326–2331.

[103] D. Cheng, C. Jiang, and X. Zhou, "Heterogeneity-aware workload placement and migration in distributed sustainable datacenters," in *Proc. IEEE 28th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2014, pp. 307–316.

[104] V. Shrivastava, P. Zerfos, K.-W. Lee, H. Jamjoom, Y.-H. Liu, and S. Banerjee, "Application-aware virtual machine migration in data centers," in *Proc. IEEE Conference on Network Communications (INFOCOM)*. IEEE, 2011, pp. 66–70.

[105] D. Huang, Y. Gao, F. Song, D. Yang, and H. Zhang, "Multi-objective virtual machine migration in virtualized data center environments," in *Proc. IEEE International Conference on Communications (ICC)*. IEEE, 2013, pp. 3699–3704.

[106] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proc. 9th ACM SIGCOMM conference on Internet measurement (IMC)*. ACM, 2009, pp. 202–208.

[107] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proc. IEEE Conference on Network Communications (INFOCOM)*. IEEE, 2010, pp. 1–9.

[108] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan, "Choreo: Network-aware task placement for cloud applications," in *Proc. 13th ACM SIGCOMM conference on Internet measurement (IMC)*. ACM, 2013, pp. 191–204.

[109] T. C. Ferreto, M. A. Netto, R. N. Calheiros, and C. A. De Rose, "Server consolidation with migration control for virtualized data centers," *Future Generation Computer Systems*, vol. 27, pp. 1027–1034, 2011.

[110] H. Jin, T. Cheocherngngarn, D. Levy, A. Smith, D. Pan, J. Liu, and N. Pissinou, "Joint host-network optimization for energy-efficient data center networking," in *Proc. IEEE 27th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2013, pp. 623–634.

[111] N. Liu, Z. Dong, and R. Rojas-Cessa, "Task and server assignment for reduction of energy consumption in datacenters," in *Proc. 11th IEEE International Symposium on Network Computing and Applications (NCA)*. IEEE, 2012, pp. 171–174.

[112] F. L. Pires and B. Báran, "Virtual machine placement literature review," *arXiv preprint*, vol. abs/1506.01509, 2015. [Online]. Available: http://arxiv.org/abs/1506.01509

[113] M. H. Ferdaus, M. Murshed, R. N. Calheiros, and R. Buyya, "Network-aware virtual machine placement and migration in cloud data centers," in *Emerging Research in Cloud Distributed Computing Systems*, S. Bagchi, Ed. IGI Global, 2015, ch. 2, pp. 42–91.

[114] Z. Usmani and S. Singh, "A survey of virtual machine placement techniques in cloud data center," *Procedia Computer Science*, vol. 78, pp. 491–498, 2016.

[115] W. Fang, X. Liang, S. Li, Chiaraviglio, and N. Xiong, "VMPlanner: Optimizing virtual machine placement and traffic flow routing to reduce network power costs in cloud data centers," *Computer Networks*, vol. 57, no. 1, pp. 179–196, 2013.

[116] T. Chen, X. Gao, and G. Chen, "Optimized virtual machine placement with traffic-aware balancing in data ceter networks," *Scientific Programming*, vol. 6, pp. 1–10, 2016, article ID 3101658.

[117] S. Fang, R. Kanagavelu, B.-S. Lee, C. H. Foh, and K. M. M. Aung, "Power-efficient virtual machine placement and migration in data centers," in *Proc. IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing.* IEEE Computer Society, 2013, pp. 1408–1413.

[118] J. Xu and J. A. Fortes, "Multi-objective virtual machine placement in virtualized data center environments," in *Proc. IEEE/ACM International Conference on Green Computing and Communications & IEEE/ACM International Conference on Cyber, Physical and Social Computing.* IEEE Computer Society, 2010, pp. 179–188.

[119] ——, "A multi-objective approach to virtual machine management in datacenters," in *Proc. 8th ACM international conference on Autonomic computing (ICAC).* ACM New York, NY, USA, 2011, pp. 225–234.

[120] Y. Gao, H. Guan, Z. Qi, Y. Hou, and L. Liu, "A multi-objective ant colony system algorithm for virtual machine placement in cloud computing," *Journal of Computer and System Sciences*, vol. 79, pp. 1230–1242, 2013.

[121] F. L. Pires and B. Báran, "Multi-objective virtual machine placement with service level agreement," in *Porc. IEEE/ACM 6th International Conference on Utility and Cloud Computing (ICAC).* IEEE Computer Society, 2013, pp. 203–210.

[122] ——, "Virtual machine placement. A multi-objective approach," in *Proc. Latin American Symposium of Infrastructure, Hardware, and Software.* IEEE, 2013, pp. 77–84.

[123] M. Ehrgott, *Multicriteria optimization.* Berlin Heidelberg: Springer-Verlag, 2006.

[124] R. K. Ahuja, T. L. Magnanti, J. B. Orlin, and M. Reddy, "Applications of network optimization," *Handbooks in Operations Research and Management Science*, vol. 7, pp. 1–83, 1995.

[125] M. Laumanns, L. Thiele, and E. Zitzler, "An adaptive scheme to generate the pareto front based on the epsilon-constraint method," in *Dagstuhl Seminar Proceedings.* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2005, article no. 04461.

[126] G. O. Inc, "Gurobi optimizer reference manual," 2015. [Online]. Available: http://www.gurobi.com

[127] H. Eriksson, "Mbone: The multicast backbone," *Communications of the ACM*, vol. 37, no. 8, pp. 54–61, 1994.

[128] B. Adamson and J. P. Macker, "Reliable messaging for tactical group communication," in *Proc. Military Communications Conference (MILCOM).* IEEE, 2010, pp. 1899–1904.

[129] S. Paul, K. K. Sabnani, J. C.-H. Lin, and S. Bhattacharyya, "Reliable Multicast Transport Protocol (RMTP)," *IEEE Journal on Selected Areas in Communications*, vol. 15, no. 3, pp. 407–421, 1997.

[130] A. Popescu, D. Constantinescu, D. Erman, and D. Ilie, "A survey of reliable multicast communication," in *Proc. 3rd EuroNGI Conference on Next Generation Internet Networks (NGI).* IEEE, 2007, pp. 111–118.

[131] J. W. Atwood, "A classification of reliable multicast protocols," *IEEE network*, vol. 18, no. 3, pp. 24–34, 2004.

[132] W. T. Strayer, B. J. Dempsey, and A. C. Weaver, *XTP: the Xpress Transfer Protocol.* Redwood City, CA: Addison-Wesley, 1992.

[133] B. Whetten, T. Montgomery, and S. Kaplan, "A high performance totally ordered multicast protocol," in *Theory and Practice in Distributed Systems.* Springer, 1995, pp. 33–57.

[134] H. W. Holbrook, S. K. Singhal, and D. R. Cheriton, "Log based receiver reliable multicast for distributed interactive simulation," *ACM SIGCOMM Computer Communication Review*, vol. 25, no. 4, pp. 328–341, Oct. 1995.

[135] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang, "A reliable multicast framework for light-weight sessions and application level framing," *ACM SIGCOMM Computer Communication Review*, vol. 25, no. 4, pp. 342–356, 1995.

[136] R. Yavatkar, J. Griffoen, and M. Sudan, "A reliable dissemination protocol for interactive collaborative applications," in *Proc. 3rd ACM international conference on Multimedia (MM)*. ACM, 1995, pp. 333–344.

[137] C. Bormann, M. J. Handley, and B. Adamson, "NACK-Oriented Reliable Multicast (NORM) Transport Protocol," RFC 5740, 2009. [Online]. Available: https://rfc-editor.org/rfc/rfc5740.txt

[138] J. P. Macker and R. B. Adamson, "A tcp friendly, rate-based mechanism for nack-oriented reliable multicast congestion control," in *Proc. IEEE Global Telecommunications Conference (GLOBECOM)*, vol. 3. IEEE, 2001, pp. 1620–1625.

[139] J. Gemmell, T. Montgomery, T. Speakman, and J. Crowcroft, "The PGM reliable multicast protocol," *IEEE network*, vol. 17, no. 1, pp. 16–22, 2003.

[140] D. Li, M. Xu, Y. Liu, X. Xie, Y. Cui, J. W. Wang, and G. Chen, "Reliable multicast in data center networks," *IEEE Transactions on Computers*, vol. 63, no. 8, pp. 2011–2024, Aug. 2014.

[141] P. Bhagwat, P. P. Mishra, and S. K. Tripathi, "Effect of topology in performance of reliable multicast communication," in *Proc. IEEE Conference on Network Communications (INFOCOM)*, vol. 2, 1994, pp. 602–609.

[142] J. Nonnenmacher and E. W. Biersack, "Reliable multicast: Where to use FEC," in *Protocols for High-Speed Networks V*, W. Dabbous and C. Diot, Eds. Springer US, 1997, ch. 4, pp. 134–148.

[143] ——, "Performance modelling of reliable multicast transmission," in *Proc. IEEE Conference on Network Communications (INFOCOM)*, vol. 2, 1997, pp. 471–479.

[144] F. Baccelli, A. Chaintreau, Z. Liu, and A. Riabov, "The one-to-manv TCP overlay: A scalable and reliable multicast architecture," in *Proc. IEEE Conference on Network Communications (INFOCOM)*, vol. 3. IEEE, 2005, pp. 1629–1640.

[145] D. Basin, K. Birman, I. Keidar, and Y. Vigfusson, "Source of instability in data center multicast," in *LADIR'10*. ACM, 2010, pp. 32–37.

[146] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk, "QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2," Internet Engineering Task Force, Internet-Draft draft-hamilton-early-deployment-quic-00, 2016, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-hamilton-early-deployment-quic-00

[147] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," *Modeling and tools for network simulation*, pp. 15–34, 2010.

[148] E. O. Elliott, "Estimates of error rates for codes on burst-noise channels," *The Bell System Technical Journal*, vol. 42, no. 5, pp. 1977–1997, 1963.

[149] G. Hasslinger and O. Hohlfeld, "The Gilbert-Elliott model for packet loss in real time services on the internet," in *Proc. 14th GI/ITG Conference on Measurement, Modelling and Evaluation of Computer and Communication Systems (MMB)*, Mar. 2008, pp. 1–15.

[150] "Internet Topology Zoo," Feb. 2017. [Online]. Available: http://www.topology-zoo.org

[151] K. Bilal, S. U. Khan, L. Zhang, H. Li, K. Hayat, S. A. Madani, N. Min-Allah, L. Wang, D. Chen, M. Iqbal *et al.*, "Quantitative comparisons of the state-of-the-art data center architectures," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1771–1783, 2013.

[152] M. Pathan and R. Buyya, "A taxonomy of CDNs," in *Content Delivery Networks*, R. Buyya, M. Pathan, and V. A., Eds. Springer, 2008, ch. 2, pp. 33–77.

[153] J. Y. Kim, G. M. Lee, and J. K. Choi, "Efficient multicast schemes using in-network caching for optimal content delivery," *IEEE Communications Letters*, vol. 17, no. 5, pp. 1048–1051, 2013.

[154] J. Ni and D. H. Tsang, "Large-scale cooperative caching and application-level multicast in multimedia content delivery networks," *IEEE Communications Magazine*, vol. 43, no. 5, pp. 98–105, 2005.

[155] B. N. Levine and J. J. Garcia-Luna-Aceves, "Improving internet multicast with routing labels," in *Proc. International Conference on Network Protocols (ICNP)*, 1997, pp. 241–250.

[156] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, vol. 47, pp. 235–256, 2002.

[157] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *arXiv preprint*, vol. abs/1402.6028, 2014. [Online]. Available: http://arxiv.org/abs/1402.6028

[158] J. Velmorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation," in *Proc. ECML'2005*, 2005, pp. 437–448.

[159] A. F. Mendelson, M. A. Zuluaga, B. F. Hutton, and S. Ourselin, "What is the distribution of the number of unique original items in a bootstrap sample?" *arXiv preprint*, vol. abs/1602.05822, 2016. [Online]. Available: http://arxiv.org/abs/1602.05822

[160] E. N. Gilbert, "Capacity of a burst-noise channel," *Bell Labs Technical Journal*, vol. 39, no. 5, pp. 1253–1265, 1960.

[161] D. Thaler and C. Hopps, "Multipath issues in unicast and multicast next-hop selection," in *Requests For Comments*. Internet Engineering Task Force, 2000, no. 2991.

[162] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein, "Maglev: A fast and reliable software network load balancer," in *Proc. 13th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2016, pp. 523–535.

[163] M. Rahman, S. Iqbal, and J. Gao, "Load balancer as a service in cloud computing," in *Proc. IEEE 8th International Symposium on Service Oriented System Engineering (SOSE)*. IEEE, 2014, pp. 204–211.

[164] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[165] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu *et al.*, "Ananta: cloud scale load balancing," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 207–218.

[166] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proc. 29th Annual ACM Symposium on Theory of Computing*. ACM, 1997, pp. 654–663.

[167] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web caching with consistent hashing," *Computer Networks*, vol. 31, no. 11, pp. 1203–1213, 1999.

[168] D. G. Thaler and C. V. Ravishankar, "Using name-based mappings to increase hit rates," *IEEE/ACM Transactions on Networking (TON)*, vol. 6, no. 1, pp. 1–14, 1998.

[169] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.

[170] R. Gandhi, Y. C. Hu, C.-K. Koh, H. H. Liu, and M. Zhang, "Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing," in *Proc. USENIX Annual Technical Conference (ATC)*, 2015, pp. 473–485.

[171] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *Proc. ACM SIGCOMM Demos*, vol. 4, no. 5, pp. 1–2, 2009.

[172] R. Wang, D. Butnariu, J. Rexford *et al.*, "Openflow-based server load balancing gone wild," *Hot-ICE*, vol. 11, p. 12, 2011.

[173] A. Aghdai, C.-Y. Chu, Y. Xu, D. H. Dai, J. Xu, and H. J. Chao, "Spotlight: Scalable transport layer load balancing for data center networks," *arXiv preprint*, vol. abs/1806.08455, 2018. [Online]. Available: https://arxiv.org/abs/1806.08455

[174] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and precise triggers in data centers," in *Proc. ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016, pp. 129–143.

[175] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Transactions on software engineering*, no. 5, pp. 662–675, 1986.

[176] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," *SIAM journal on computing*, vol. 29, no. 1, pp. 180–200, 1999.

[177] H. Shen and C.-Z. Xu, "Locality-aware and churn-resilient load-balancing algorithms in structured peer-to-peer networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 849–862, 2007.

[178] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[179] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones," in *Proc. 10th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, vol. 13, 2013, pp. 185–198.

[180] V. Cardellini, M. Colajanni, and S. Y. Philip, "Dynamic load balancing on web-server systems," *IEEE Internet computing*, vol. 3, no. 3, p. 28, 1999.

[181] Q. Zhang, L. Cherkasova, and E. Smirni, "Flexsplit: A workload-aware, adaptive load balancing strategy for media clusters," in *Electronic Imaging 2006*, vol. 60710. International Society for Optics and Photonics, 2006, p. 60710I.

[182] G. Ciardo, A. Riska, and E. Smirni, "EquiLoad: a load balancing policy for clustered web servers," *Performance Evaluation*, vol. 46, no. 2, pp. 101–124, 2001.

[183] Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo, "Workload-aware load balancing for clustered web servers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 3, pp. 219–233, 2005.

[184] S. Sharifian, S. A. Motamedi, and M. K. Akbari, "A content-based load balancing algorithm with admission control for cluster web servers," *Future Generation Computer Systems*, vol. 24, no. 8, pp. 775–787, 2008.

[185] "HAProxy: the reliable, high-performance TCP/HTTP load balancer." [Online]. Available: http://www.haproxy.org

[186] R. Gandhi, Y. C. Hu, and M. Zhang, "Yoda: A highly available layer-7 load balancer," in *Proc. 11th European Conference on Computer Systems*. ACM, 2016, article no. 21.

[187] F. William, *An introduction to probability theory and its applications*. New York: John Wiley And Sons Inc., 1950.

[188] D. J. Newman, "The double dixie cup problem," *The American Mathematical Monthly*, vol. 67, no. 1, pp. 58–61, 1960.

[189] D. S. E. Deering and B. Hinden, "IP Version 6 Addressing Architecture," RFC 4291, Feb. 2006. [Online]. Available: https://rfc-editor.org/rfc/rfc4291.txt

[190] M. D. Mitzenmacher, "The power of two choices in randomized load balancing," Ph.D. dissertation, University of California at Berkeley, 1996.

[191] J. D. Little, "A proof for the queuing formula: $L = \lambda W$," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[192] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system.* Hudson, MA: Eastern Research Laboratory, Digital Equipment Corporation, 1984, vol. 38.

[193] "The Apache HTTP server project." [Online]. Available: http://www.apache.org

[194] E.-J. van Baaren, "Wikibench: A distributed, Wikipedia based web application benchmark," Master's thesis, VU University Amsterdam, 2009.

[195] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, vol. 53, no. 11, pp. 1830–1845, Jul. 2009.

[196] Y. Desmouceaux, M. Enguehard, V. Nguyen, P. Pfister, W. Shao, and E. Vyncke, "A content-aware data-plane for efficient and scalable video delivery," in *Proc. 16th IFIP/IEEE Symposium on Integrated Network and Service Management (IM).* IEEE, 2019, pp. 10–18.

[197] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2011, pp. 295–308.

[198] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. ACM Conference of the Special Interest Group on Data Communication (SIGCOMM).* ACM, 2017, pp. 15–28.

[199] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with Beamer," in *Proc. 15th USENIX Symposium on Networked Systems Design & Implementation (NSDI).* USENIX Association, 2018, pp. 125–139.

[200] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[201] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014.

[202] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4 → NetFPGA workflow for line-rate packet processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, 2019, pp. 1–9.

[203] F. Duchene and O. Bonaventure, "Making multipath TCP friendlier to load balancers and anycast," in *Proc. IEEE 25th International Conference on Network Protocols (ICNP).* IEEE, 2017, pp. 1–10.

[204] A. Medina, M. Allman, and S. Floyd, "Measuring interactions between transport protocols and middleboxes," in *Proc. 4th ACM SIGCOMM conference on Internet measurement (IMC).* ACM, 2004, pp. 336–341.

[205] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend TCP?" in *Proc. 11th ACM SIGCOMM conference on Internet measurement (IMC).* ACM, 2011, pp. 181–194.

[206] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 123–137.

[207] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: state of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2018.

[208] A. Gandhi, S. Doroudi, M. Harchol-Balter, and A. Scheller-Wolf, "Exact analysis of the M/M/k/setup class of Markov chains via recursive renewal reward," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1. ACM, 2013, pp. 153–166.

[209] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi, "Integrated and autonomic cloud resource scaling," in *Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS)*. IEEE, 2012, pp. 1327–1334.

[210] A. Beloglazov and R. Buyya, "Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers," in *Proc. 8th International Workshop on Middleware for Grids, Clouds and e-Science*. ACM, 2010, pp. 4:1–4:6.

[211] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "Dejavu: accelerating resource allocation in virtualized environments," in *ACM SIGARCH computer architecture news*, vol. 40, no. 1. ACM, 2012, pp. 423–436.

[212] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters," in *Proc. 6th international conference on Autonomic computing*. ACM, 2009, pp. 117–126.

[213] J. Lioris, R. Pedarsani, F. Y. Tascikaraoglu, and P. Varaiya, "Platoons of connected vehicles can double throughput in urban roads," *Transportation Research Part C: Emerging Technologies*, vol. 77, pp. 292–305, 2017.

[214] K. Lee, R. Pedarsani, and K. Ramchandran, "On scheduling redundant requests with cancellation overheads," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1279–1290, 2017.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Protocoles de couche réseau pour l'extensibilité des centres de données

Mots clés : Réseaux de centres de données, Mobilité, Multicast, Équilibre de charge, Segment Routing

Résumé : Du fait de la croissance de la demande en ressources de calcul, les architectures de centres de données gagnent en taille et complexité. Dès lors, cette thèse prend du recul par rapport aux architectures réseaux traditionnelles, et montre que fournir des primitives génériques directement à la couche réseau permet d'améliorer l'utilisation des ressources, et de diminuer le trafic réseau et le surcoût administratif. Deux architectures réseaux avancées, Segment Routing (SR) et Bit-Indexed Explicit Replication (BIER), sont utilisées pour construire et analyser des protocoles de couche réseau, afin de fournir trois primitives : (1) mobilité des tâches, (2) distribution fiable de contenu, et (3) équilibre de charge.
Premièrement, pour la mobilité des tâches, SR est utilisé pour fournir un service de migration de machine virtuelles sans perte. Cela ouvre l'opportunité d'étudier comment orchestrer le placement et la migration de tâches afin de (i) maximiser le débit inter-tâches, tout en (ii) maximisant le nombre de nouvelles tâches placées, mais (iii) minimisant le nombre de tâches migrées. Deuxièmement, pour la distribution fiable de contenu, BIER est utilisé pour fournir un protocole de multicast fiable, dans lequel les retransmissions de paquets perdus sont cibleés vers l'ensemble précis de destinations n'ayant pas reçu ce paquet : ainsi, le surcoût de trafic est minimisé. Pour diminuer la charge sur la source, cette approche est étendue en rendant possibles des retransmissions par des pairs locaux, utilisant SR pour trouver un pair capable de retransmettre. Troisièmement, pour l'équilibre de charge, SR est utilisé pour distribuer des requêtes à travers plusieurs applications candidates, chacune prenant une décision locale pour accepter ou non ces requêtes, fournissant ainsi une meilleure équité de répartition comparé aux approches centralisées. La faisabilité d'une implémentation matérielle de cette approche est étudiée, et une solution (utilisant des canaux cachés pour transporter de façon invisible de l'information vers l'équilibreur) est implémentée pour une carte réseau programmable de dernière génération. Finalement, la possibilité de fournir de l'équilibrage automatique comme service réseau est étudiée : en faisant passer (avec SR) des requêtes à travers une chaîne fixée d'applications, l'équilibrage est initié par la dernière instance, selon son état local.

Title: Network-Layer Protocols for Data Center Scalability

Keywords: Data-center networking, Task Mobility, Multicast, Load Balancing, Segment Routing

Abstract: With the development of demand for computing resources, data center architectures are growing both in scale and in complexity. In this context, this thesis takes a step back as compared to traditional network approaches, and shows that providing generic primitives directly within the network layer is a great way to improve efficiency of resource usage, and decrease network traffic and management overhead. Using two advanced network architectures, Segment Routing (SR) and Bit-Indexed Explicit Replication (BIER), network layer protocols are designed and analyzed to provide three high-level functions: (1) task mobility, (2) reliable content distribution and (3) load-balancing.
First, task mobility is achieved by using SR to provide a zero-loss virtual machine migration service. This then opens the opportunity for studying how to orchestrate task placement and migration while aiming at (i) maximizing the inter-task throughput, while (ii) maximizing the number of newly-placed tasks, but (iii) minimizing the number of tasks to be migrated. Second, reliable content distribution is achieved by using BIER to provide a reliable multicast protocol, in which retransmissions of lost packets are targeted towards the precise set of destinations having missed that packet, thus incurring a minimal traffic overhead. To decrease the load on the source link, this is then extended to enable retransmissions by local peers from the same group, with SR as a helper to find a suitable retransmission candidate. Third, load-balancing is achieved by way of using SR to distribute queries through several application candidates, each of which taking local decisions as to whether to accept those, thus achieving better fairness as compared to centralized approaches. The feasibility of hardware implementation of this approach is investigated, and a solution using covert channels to transparently convey information to the load-balancer is implemented for a state-of-the-art programmable network card. Finally, the possibility of providing autoscaling as a network service is investigated: by letting queries go through a fixed chain of applications using SR, autoscaling is triggered by the last instance, depending on its local state.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France