Two challenges of Software Networking: Name-based Forwarding and Table Verification

Leonardo Linguaglossa

UNIVERSITÉ SORBONNE PARIS CITÉ

UNIVERSITÉ PARIS DIDEROT

ÉCOLE DOCTORALE : Sciences Mathématiques de Paris Centre
Laboratoire : IRIF
en collaboration avec : Nokia Bell Labs, INRIA

DOCTORAT
Informatique

AUTEUR : Leonardo Linguaglossa

Two challenges of Software Networking: Name-based Forwarding and Table Verification

Deux défis des Réseaux Logiciels : Relayage par le Nom et Vérification des Tables

Thèse dirigée par : Fabien Mathieu, Diego Perino, Laurent Viennot

Soutenue le 09/09/2016

JURY

M. Walid Dabbous	Rapporteur
M. Frédéric Giroire	Examinateur
M. Fabien Mathieu	Encadrant
M. Diego Perino	Encadrant
Mme Maria Potop-Butucaru	Examinateur
M. Dario Rossi	Examinateur
M. Gwendal Simon	Rapporteur
M. Laurent Viennot	Directeur de thèse

Acknowledgements

I would like to express to:

- *everyone: many thanks.*

Contents

Résumé en langue française

Dans ce travail de thèse, nous identifions deux principaux aspects de l'évolution de l'Internet: un aspect "comportemental", qui se réfère à un changement survenu dans le type d'interaction entre les utilisateurs et le réseau, et un aspect "structurel", lié au problème de l'évolution Internet d'un point de vue architectural.

Le premier aspect est décrit comme un décalage entre l'utilisation du réseau et les fonctions réelles qu'il fournit. Alors que le réseau met en œuvre les simples primitives d'envoi et de réception de paquets génériques, les utilisateurs sont vraiment intéressés par l'extraction et consommation des contenus.
Les communautés scientifiques et industrielles ont proposé des solutions pour la transformation de l'Internet vers un réseau des contenus : les CDNs (Content Delivery Networks) sont une type de réalisations de réseau des contenu au niveau utilisateur. Ce type de solution fournit un bon service des échange des contenus, au prix d'une efficace réduite.
Plusieurs solutions au niveau Réseau ont été proposé comme alternatif aux implémentations utilisateur. Par contre, une réalisation de ce type de paradigme exige une réinitialisation complète des réseaux : mise à jours des dispositifs, nouveau software, et configurations.
Nous proposons l'utilisation des fonctionnalités avancées basés sur les contenus sans adoption d'une approche de type « clean-slate ». La base pour notre proposition est NDN (Named-data Networking). « Caesar », est notre proposition pour un routeur qui essaye d'avancer l'état de l'art par la mise en œuvre des fonctionnalités basées sur le contenu qui peuvent coexister avec des environnements réels du réseau.
Cette architecture est réalisable et les mises à jour peut être entièrement compatible avec les réseaux existants.

Le problème de la lente évolution de l'infrastructure de l'Internet réside encore plus dans sa conception architecturale, qui a été montré être faiblement extensible. Tandis-que les applications utilisateurs ont profités des avancement software, et les moyenne des transport de l'information ont été améliorés par les avancements technologiques, le « centre » de l'Internet, le protocole IP, est resté toujours le même protocole proposé dans les années '70. SDN (Software-Defined Networking) est un type de paradigme réseau qui adresse ce problème en proposant la séparation entre le plan contrôle et donnés : cette séparation montre une flexibilité élevé par rapport à l'architecture Internet courant. Un réseau SDN peux contenir plusieurs type de protocoles, et peux être mise à jour très facilement.
Le coût de SDN est relatif surtout à la gestion des erreurs. Par exemple, la détection des boucles dans le réseau devient un problème challenging.
Une inspection complète du réseau peut être infaisable, selon la taille du réseau donné, sa typologie ou le pattern du trafic. « SDN verification » est une nouvelle direction de recherche dont le but est de vérifier la cohérence et la sécurité des configurations de réseau en fournissant une validation formelle ou empirique.
Nous proposons un outil mathématique pour la détection des boucles qui prend en compte

l'analyse des tables de forwarding.

Le manuscrit est composé d'un chapitre d'Introduction, qui décrit les base pour une correcte compréhension des challenges et des résultats de ce travail de thèse.
Il est suivi par un chapitre de « Background » (Chapitre 1) qui contient la description de l'état de l'art sur les deux aspects mentionnés. Les deux nouveaux paradigmes de réseaux utilisés comme base pour cette thèse sont décrits en detaille : il s'agit de NDN et SDN. La thèse continue avec les deux parties qui correspondent « amélioration du plan des données » et « vérification des tables de routage ».
Dans la première partie nous proposons « Caesar » (Chapitre 2), un dispositif réseau capable d'adresser le problème de la distribution des contenus en utilisant des primitives de forwarding basés sur le nom des donnés et non pas sur l'adresse IP. Caesar est présenté dans deux chapitre, qui décrivent l'architecture et les principaux modules : Forwarding (Chapitre 3) et Gestion des Requêtes (Chapitre 4).
Nous presentons le problème de la détection des boucles (Chapitre 5) et proposons dans la deuxième partie un outil mathématique pour la vérification efficace des boucles dans le réseau SDN d'un point de vue théorique, en discutant les améliorations par rapport à l'état de l'art (Chapitre 6). Nous concluons cette thèse avec un résumé des principaux résultats obtenus dans ces deux parties, suivi de la présentation des travaux en cours et futurs.

Introduction

Since the beginning, the Internet changed the lives of network users similarly to what the telephone invention did at the beginning of the 20th century. While Internet is affecting users' habits, it is also increasingly being shaped by network users' behavior (cf. Background Section 1.3, page 12). Several new services have been introduced during the past decades (i.e. file sharing, video streaming, cloud computing) to meet users' expectation. The Internet is not anymore a simple network meant to connect nodes providing few websites access: this influences the network traffic pattern and the users' network usage. As a consequence, although the Internet infrastructure provides a good best-effort service to exchange information in a point-to-point fashion, this is not the principal need that todays users request. Current networks necessitate some major architectural changes in order to follow the upcoming requirements, but the experience of the past decades shows that bringing new features to the existing infrastructure may be slow (a well known example is the IPv6 protocol, defined in the late Nineties and slowly spreading only in the last few years).

In the current thesis work, we identify two main aspects of the Internet evolution: a "behavioral" aspect, which refers to a change occurred in the way users interact with the network, and a "structural" aspect, related to the evolution problem from an architectural point of view. The behavioral perspective states that there is a mis-match between the usage of the network and the actual functions it provides. While network devices implements the simple primitives of sending and receiving generic packets, users are really interested in different primitives, such as retrieving or consuming content. The structural perspective suggests that the problem of the slow evolution of the Internet infrastructure lies in its architectural design, that has been shown to be hardly upgradeable.

On the one hand, to target the new network usage, the research community proposed the Named-data networking paradigm (NDN), which brings the content-based functionalities to network devices. On the other hand Software-defined networking (SDN) can be adopted to simplify the architectural evolution and shorten the upgrade-time thanks to its centralized software control plane, at the cost of a higher network complexity, that can easily introduce

some bugs. Both NDN and SDN are two novel paradigms which aim to innovate current network's infrastructure, but despite sharing similar goals, they act at different levels.

The rationale behind NDN comes from the observation of current Internet usage. Nowadays, users send emails, use chats and surf the Web no more than sharing multimedia files or watching YouTube videos. Modern Internet is in fact a *content network*, where users are interested in retrieving and consuming some content. Every piece of information flowing in the Internet can be associated to a content: from the single web page to the specific packet of an on-line video stream. Unfortunately, the network infrastructure is not suited for this new traffic patterns, being essentially the same architecture as the one introduced in the Seventies. To meet the new changes in the network usage, the research community recently proposed the NDN paradigm (cf. Background Section 1.4, page 15). NDN proposes to enrich the current network with content-based functionalities, that can improve both network efficiency and users' experience. NDN requires a change in the network architecture to be fully deployed. The first question that we want to answer is: are we able to exploit those advanced functionalities without adopting a clean-slate approach? We address this question by designing and evaluating Caesar, a content router that advances the state-of-the-art by implementing content-based functionalities which may coexist with real network environments. We would like to prove that upgrading existing architecture is feasible and that upgrades can be fully compatible with existing networks.

The SDN proposal tackles another side of the evolution problem, and its main purpose is to deal with the Internet's slow adaptivity to changes. In fact, reacting to new paradigms implies that several modifications have to be done in network devices, their interconnections, implementation and in the final deployment. These changes usually involve hardware devices, whose main task is unique and fixed by the original design. Thus, adding new services usually translates into buying new components. SDN (cf. Background Section 1.5, page 25) is expected to dramatically reduce the time required to configure a whole network. SDN proposes a network architecture made of two separate *planes*: a control plane, centralized and software programmable, which defines the "behavior" of the network, and a data plane, consisting of a set of network devices which are managed by the control plane and are responsible for the actual data processing. SDN networks are becoming increasingly popular thanks to their ease of management and their flexibility: changing the network's behavior requires to simply program a software controller, allowing the network architecture to evolve at software speed. Moreover, the actual behavior of the network may be defined after it has been deployed, without replacing any network device, completely under the control of the network owner. Despite its ease of management, SDN may generate data misconfiguration on the managed network, with consequent broken connectivity, forwarding loops, or access control violations. A full network inspection may be unfeasible, depending on the size of the given network, its typology or the traffic pattern. SDN verification is a novel research direction which aims to check the consistency and safety

of network configurations by providing formal or empirical validation. SDN verification tools usually take as input the network topology and the forwarding tables, to verify the presence of problems such as packets creating a loop. Hence, the second question we want to answer is: can we detect forwarding loops in real time? We direct our efforts toward network misconfiguration diagnosis by means of analysis of the network topology and forwarding tables, targeting the problem of detecting all loops at real-time and in real network environments.

Thesis organization

This thesis work is organized in three parts, with an introductory background chapter and two main parts.

The background serves as a survey of current network architecture, and to introduce the main trends in the evolution of the Internet. This chapter also presents the motivations of the work of the subsequent parts.

The first part of this thesis is devoted to the *data plane enhancement*, with focus on the integration of content-based functionalities in actual network equipment. It describes Caesar, our prototype of an advanced network device that is capable of performing content-based NDN operations. Caesar is fully compatible with the state-of-the-art Internet routers, while adding advanced functionalities depending on the traffic type it receives. We show that Caesar's forwarding performance are comparable with high-speed edge routers, and can sustain a rate of several millions of packets per second.

The second part of this thesis considers the problem of network diagnosis in the environment of software-defined networks. The ease of management and customization in SDN comes at the cost of a more difficult detection of network configuration bugs. We solve this issue proposing a theoretical framework which allows us to prove that the problem of network verification can be bounded for practical networks. Our framework consists of a mathematical model of forwarding networks and some algorithms which can be used to detect forwarding loops.

We conclude this thesis providing a summary of the main results achieved, and presenting ongoing and future works. For a greater ease of reading, a glossary can be found at page 144, while tables of symbols are shown at page 103 and 137.

Publications

The content of Chapter 3 is published in [PVL+14b], while the results of Chapter 4 have been partially published in [VPL13]. Preliminary results of Chapter 6 are presented in [BDLPL+15], while another publication is currently under submission for Chapter 6.

The patent [PVL14a], filed in 2014, contains the description of the algorithm presented in Chapter 3.

A complementary demo, whose description is published in [PGB+14], demonstrates the feasibility of a *content-aware* network in the mobile back-haul setting, leveraging Caesar's design, and show NDN provides significant benefits for both user experience and network cost in this scenario. This is out of the main scope of this thesis and therefore not reported.

Chapter 1

Background

This section serves as background for a better understanding of this thesis. Specifically, we introduce some basic definitions in the field of Computer Networks, with particular emphasis on the Internet and its history. We then focus on the evolution of the Internet and the importance of an evolvable network architecture. Finally, we introduce our approach and contributions in the areas of **Information Centric Networking** and **Software-Defined Networking**, which are two key technologies for the evolution of the current Internet architecture. When acronyms are not purposely explained in the text, a description can be found in the glossary at page 144.

1.1 Telecommunication Networks

A telecommunication network is the interconnection of two or more network devices, or nodes, which are able to transport information between endpoints. Two examples of such a network are the telephone network and the Internet. A network can be mathematically modeled as a *graph*. Let V be the set of network nodes; let E be the set of edges, or the links between nodes; then the graph $G = (V, E)$ is the graph that represents the network.

While networks were classified in the past in two main categories (analog and digital networks) today all networks are digital: the information (voice, data, video) is encoded into bits and then translated to electrical signals. The signals are transmitted through a transmission medium (e.g. wires, for a wired network; radio waves, for wireless devices) and are regenerated when reaching another network device, or decoded at the final destination. All communication channels may introduce some errors due to noise that can cause wrong bit decoding. Shannon's theory of communication [Sha01] demonstrates that the probability of errors for a digital transmission

over a noisy channel can be tuned to be almost error-free. There are other causes of errors that are not related to the channel noise (cf. Section 1.1.1); anyway, error-recovery mechanisms can be implemented to avoid such these errors. For a detailed description about error-recovery techniques and data-loss avoidance, refer to the survey [LMEZG97] and RFC 2581 [SAP99].

1.1.1 Circuit switching and packet switching

There exist two methodologies of implementing a telecommunication network: *circuit switching* and *packet switching*.

In a *circuit-switched network*, a user[1] is identified by a unique ID (e.g. the telephone number). A physical communication channel is established between endpoints (for instance, two users) in order to communicate. For instance, this is the case of the traditional telephone network, where every call between two users requires an ad-hoc communication circuit. As a consequence, when the circuit is built, the two users are normally unavailable for other communications[2]. A communication channel may introduce errors due to thermal noise, On the one hand, circuit switching is a reliable transport for information as long as the circuit exists: the transmission is generally error-free, because a channel exists between the two callers and it is reserved for that specific call; in addition, the call is real-time, and the data transmitted does not require complex ordering mechanisms. On the other hand, without a circuit no communication is possible, implying that the usage of the resources is not optimal: i.e. when a channel is reserved but the information is transmitted only during a fraction of the call duration. Some special devices, known as *telephone exchanges* store the information about how to reach any user in some area, and how to create a circuit for the communications.

In a *packet-switched network* every network node is identified by an ID called address. Following a different approach, such a network does not create a communication path between endpoints, but relies on groups of data called packets. Every packet is composed of two parts: the **header** and the **payload**: the former contains the information to reach the destination, while the latter contains the bits of the information which is to be transmitted. A packet may require to traverse several intermediate nodes (also known as *hops*) before being delivered at the final destination. It is therefore required for every node to know the next hop where any packet should be sent to reach its destination. We call **routing** the process of finding one or more paths in the network for the specific destination stored in the packet header. Each packet is routed independently: a routing algorithm is the part of the routing responsible for deciding which output line an

[1] The term *user* identifies here a physical person, but we could also use it as a metonymy for an application, a device, or any actor of the communication.

[2] This is not the case of today's telephony, where the voice is transmitted using the Internet as a backbone and multiple calls may be held even if the line is busy: in France almost 95% of telephony is Internet-based [Ltd13].

incoming packet should be transmitted on [Tan96, p. 362]. The route computation can be manually-configured (*static routing*) when paths are installed by the network administrator at every node, as opposed to *dynamic routing*, in which specific algorithms are performed (generally in a distributed manner) to gather information about the network topology and building paths accordingly. The routing algorithms usually calculate the network topology, or a snapshot of it; then they store the interface(s) of the next-hop(s) for all possible destinations in a forwarding table; as soon as an incoming packet arrives, a lookup is performed and the packet is forwarded to the next-hop. Common routing algorithms can efficiently calculate "optimal" paths according to a metric: for instance shortest path algorithms compute the routes that minimize the length of the tranmission path, in terms of number of hops or geographical distance, while other algorithms may optimize any given cost function. Two typical algorithms for shortest paths calculation are the Bellman-Ford algorithm [Bel56, For56] and the Dijkstra algorithm [Dij59]. Since a routing algorithm specifies how routes are chosen, different algorithms may provide different routes for the same pair of nodes. The routing protocol is the process of disseminating the route information discovered by the routing algorithm. It defines as well the format of the information exchanged between nodes and specifies how forwarding tables are calculated. Finally, it encapsulates route information in regular packets, to be sent to the neighbor nodes Since the network topology may change over time, the routing process continuously sends route updates to all the routers with the current topology changes.

Some examples of routing protocols are Open Shortest Path First (OSPF[3]) and Routing Information Protocol (RIP[3]), using respectively the Dijkstra's and the Bellman-Ford's shortest path algorithms, and the Border Gateway Protocol (BGP[3]) protocol, in which the policy (and so the choice of the paths) depends on service level agreements between network providers. As a result of the routing process, all nodes have a *routing table* which keeps track of the destination where a packet should be sent to reach a target node. For a source node there can be multiple possible paths that conduct to the same destination, and vice-versa there may be multiple locations reachable from the same next-hop. Such a network allows to have shared communication channels (i.e. multiple communications on the same wire) and it is flexible: it does not need circuits to start communication. However in a packet-switched network the delivery is not granted, because some packets may be lost along the path towards the destination node in consequence of a link collapse or an intermediate node failure. The incorrect transmissions may be compensated by error-detection/correction mechanisms (e.g. in the case of faulty header or payload). On the occurrence of a link failure, the routing protocol shall be able to detect another path (if any) for that destination node. Being natively **connectionless**, a packet-switched network can also emulate the behavior of a circuit-switched network using virtual circuit-switching.

[3]For more details about OSPF, RIP and BGP protocols, refer to the Glossary at page 144.

1.1.2 Computer Networks

Computer Networks are a subset of telecommunication networks. In this case, network nodes are computers, and the network can provide some additional services beyond the simple information transport, with the World Wide Web, digital audio/video and storage services being some examples. When computers interact there are mainly two possible models of interactions: **push** and **pull**. *Push communication* is a communication that is initiated by a sender towards the receivers. The actors of this model are the information producer, or the *Publisher*, and the consumer, or the *Subscriber*. The subscriber signs up for the reception of some data, that will be "pushed" by the publisher when ready. This mechanism is called *Publish/Subscribe*. An example of service based on Publish/Subscribe model is the mailing list service: a user can subscribe to a mailing list to receive any e-mail sent to the mail address of the list at anytime; then he can asynchronously consume the e-mails when needed.
Conversely, *Pull communication* is a communication method in which the receiver actively requests some information, and the sender reacts to this request by sending back the requested data. Pull communication is the base of the *Client/Server model*: a server executes a wait procedure until it receives a request from a user; then it performs a processing routine and sends back what the user has demanded.

The best-known example of a packet-switched computer network is the Internet, that was initially based on a pull-communication model. The next section briefly shows its history and then it focuses on the Internet's architecture.

1.2 The Internet

The Internet is a telecommunication network, and in particular a computer network, that was born as an evolution of the former *ARPANET* network, between the Sixties and the Seventies. The main goal was to interconnect few nodes, located in different areas in the USA, to implement some basic resource-sharing, text and file transfer. At the beginning, it only acted as an improved alternative to the global telephone network for the voice transmission [Tan96, p. 55].

As in the client/server model, two entities were involved in the communication: a client, that requested some resource, and a server, that provided the requested resource (e.g. data, computational power, storage). At that time there were many client terminals which asked for resources, and few main computers capable of providing them. User terminals communicated with the servers knowing the servers' addresses, that were related to their geographical posi-

Internet Layer	Protocols	OSI Number
Application	Web, Emails, DNS	7
Transport	TCP, Congestion control, UDP	4
Internet	IPv4, IPv6, ICMP	3
Link	Ethernet, Bluetooth, WiFi	1/2

Figure 1.1: *Internet and OSI layers compared, with some protocols as example.*

tion. Today's Internet is still built upon this design choice, and the network infrastructure is location-based, or **host-centric**.

1.2.1 The TCP/IP Protocol Stack

The Internet architecture is logically divided in multiple **layers**. Each layer is responsible for a certain set of functions, and can exchange information with the contiguous layers of the same node, or with the homologous layer on another node. The ISO-OSI model [Zim80] represents the *de jure* standard for the definition of each layer and its services. Despite this standardization, proposed at the end of the Seventies, the TCP/IP [CK74] is nowadays the *de facto* standard. It is a protocol suite for the Internet, so-called because of the two main protocols: Internet Protocol and Transmission Control Protocol. We will assume the TCP/IP as our model for the remaining of the thesis.

Layers are traditionally represented in a vertical fashion, as shown in Figure 1.1. The bottom layers are close to the physical transmission of electrical signals, while the top layers interact with the users. In the following we provide a bottom-up overview of the Internet Layers.

The *Link Layer* is responsible for the transmission and the reception of electrical signals. Moreover it transforms the signals received to bits (and vice versa) for the higher layer (respectively, from the higher layer). A sublayer of the Link layer is the *Medium Access Control*, which is responsible of assembling the bits into packets (called *frames* at this level). It is also responsible of transmitting frames between two physically connected devices. This operation is called **forwarding**.
The *Internet Layer*, or *Network Layer*, is responsible for connecting two nodes which are not physically connected. It calculates the network layer topology by running a **routing protocol**. IP is the main Network layer protocol, which defines the format of the header and the payload. To forward packets it performs a lookup in a routing table using IP addresses as key, in order to find the next connected hop that can be exploited to reach the final packet destination. It also assembles the packet (called *datagram* at this level) for the Link Layer.
The *Transport Layer* is responsible of creating a communication channel between the endpoints.

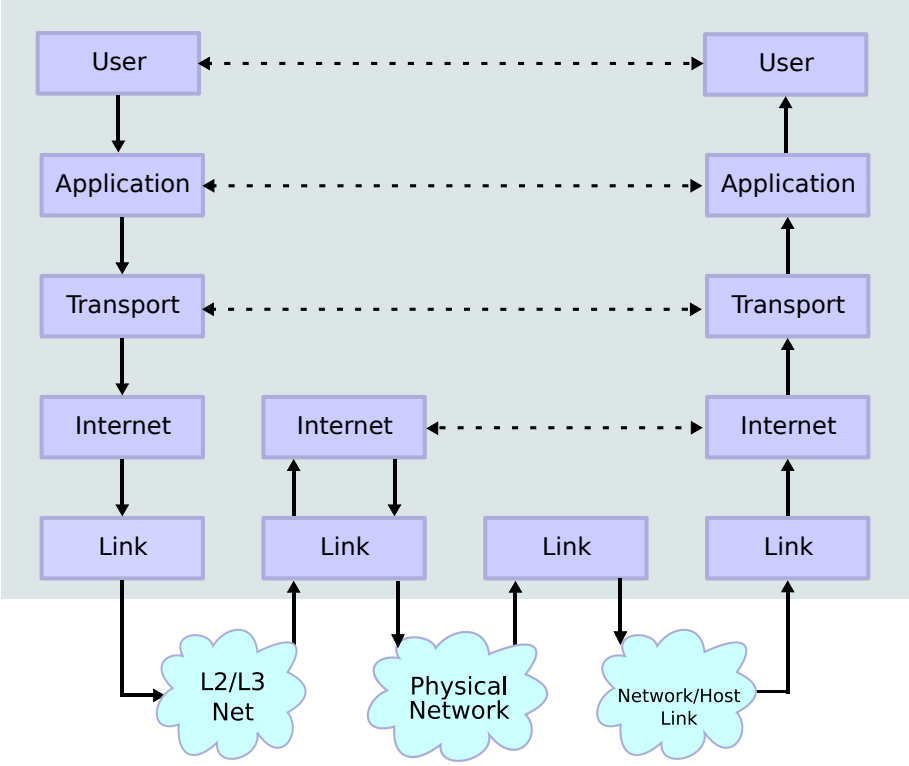


Figure 1.2: *The Internet protocol stack. Each layer abstracts a specific network function, and can communicate with its homologous layer on the other endpoint or with the adjacent layers through standard interfaces.*

The TCP protocol is executed in order to create a reliable connection-oriented communication over a connectionless channel. It performs a congestion control algorithm to shape the traffic in the network and avoid to overload the network. The TCP packets are also called *segment* at this level. Whenever either a reliable channel or a congestion-control mechanism is not needed, the UDP protocol may be used to implement the most simple best-effort delivery mechanism. Finally, the *Application Layer* is the one in which a user can use the resource requested, and implement applications as e-mails or web browsing. An example of a function implemented at the Application layer is the Domain Name System (DNS), a service that matches human-readable strings such as `www.inria.fr` to the corresponding IP address of a server.

Figure 1.2 shows the Internet protocol stack and a typical interaction between layers and nodes. Users represent the endpoints of the communication, and they usually create requests or data at the application layer. The packet created is then passed among users in a top-down manner. Every layer encapsulates packets adding the header needed by the current layer. At the bottom layer, the packet is transmitted to a neighbor, which may ascend the protocol stack decapsulating the packet. To clarify this idea, a L2 device implements the Link Layer only, while a L3 device (a router) decapsulates the packet up to the Internet Layer. Finally end-nodes decapsulate packets up to the Application layer.

Plane	Protocols
Data Plane	MAC forwarding, File transfers, Skype call[a]
Control Plane	ICMP, OSPF, ARP, Skype signaling

[a] Skype data packets.

Figure 1.3: *Data and Control Plane.*

1.2.2 Data Plane and Control Plane

We can further classify the architecture into the **Data Plane** and the **Control Plane** [YDAG04]. The *Data Plane*, also called *Forwarding Plane*, carries the data traffic Data Plane tasks are time-critical, and can be classified as follows:

- **Packet input**, which includes all the operations a node performs to receive a packet;
- **Packet processing**, which consists of all the operations performed on the current packet, e.g. the packet classification and the lookup in a forwarding table to get the next hop;
- **Packet output**, which includes all the operations related to the transmission of a packet to the following node.

All nodes have a Data Plane and different elements performing Data Plane operations. As an example, in a router **Line Cards** are in charge of Data Plane processing. High-performance IP routers may be equipped with several line cards performing high-speed operations.

The *Control Plane* is the part of the network architecture configuring the traffic flow in the network. The control plane instructs the network elements on how to forward packets. Routing protocols belong to the control plane, even if control packets are transmitted in the Data Plane. Control plane tasks are not real-time, and can be performed on a longer time scale compared to the Data Plane. They mainly consist of route dissemination, table maintenance and traffic control.

The name "planes" is inspired by the lack of distinction with respect to the layers in which a certain control or data plane operation take place: any of the elements and the protocols of the TCP/IP protocol stack may be part of the Data or the Control Plane, according to the function that they implement. In other words, planes are "orthogonal" to layers. An overview of examples of Data and Control Plane functions is shown in Figure 1.3.

1.3 The evolution of the Internet

The usage of the Internet changed with the creation of new services (1971, e-mail service; 1991, World Wide Web; 2002, BitTorrent; 2005, YouTube; 2008, Dropbox) and the introduction of new technologies (1992, multimedia compression, i.e. MPEG[4]; 1999, low cost RFIDs[5] and wireless technologies standardization). Furthermore, an increasing number of users have now access to the Web, and this world-wide diffusion generated a boost in the network traffic and, most of all, originated a different paradigm of network usage.

To investigate the evolution of the Internet and its usage, we focus on two main aspects of the problem. The first one is "behavioral", and refers to a change occurred in the way users interact with the network: there is a mis-match between the use of the network and the functions provided, or, from a different perspective, the infrastructure does not fit with users' behavior The second one is "structural", and refers to the evolution problem from an architectural point of view. In other words, not only the use of the network is different from its original design, but the architecture's design itself is difficult to upgrade and improve.
In the following, we first describe the behavioral change of the Internet usage, which lead to the definition of a new paradigm called Information Centric Networking (ICN); we show that it is one of the approaches proposed to solve the behavioral issue (for a detailed description, see Section 1.4). Then we target the structural problem, and we show that the paradigm of Software-Defined Networking (SDN) could be a solution to the architectural evolution problem (SDN is described in Section 1.5).

1.3.1 The behavioral issue

The behavioral change occurred as a consequence to the features provided by new technologies, which slightly shifted the users' paradigm of communication. Already in 2009 authors of [MPIA09] point out that more than 70% of the residential Internet traffic was made of HTTP requests or P2P sharing protocols (eDonkey, BitTorrent), with more than one third of the HTTP traffic consisting of video downloads/uploads; nowadays the ratio is even bigger. The massive popularity of streaming platforms such as YouTube, and the diffusion of user-generated content led to a paradigm that is no more location-based, but rather content-based. This **content-oriented** model is centered on the content itself: users want to retrieve information regardless of the exact location of the server that provides it. We describe now why building such a mechanism over an IP underlay is not efficient [rg16].

[4] Acronym for Moving Picture Experts Group. See the glossary at page 144 for more details.
[5] Acronym for Radio Frequency Identifiers. See the glossary at page 144 for more details.

First of all, the conversational nature of Internet gives emphasis to the endpoints: IP addresses at network layer represent source and destination endpoints, while a content-oriented architecture requires a generalization to map any information to a name. Users are increasingly using mobile platforms to access Internet's content, and even though some effort has been done to make mobility transparent to applications and higher level protocols [Per98], IP does not support it natively. Finally, IP lacks in the support of secure communication: protocols such as SSL contribute to build a secure channel between endpoints, but a per-content security may be required when moving towards a content-oriented network.

The need for a novel communication model inspired both research community and industrial R&D, so that a few solutions to match the new traffic pattern have been proposed. It is the case of the content-delivery networks (CDN): they are global networks that provide content-distribution mechanisms built at the Application layer on top of the current Internet infrastructure, representing the state-of-the-art implementation of a content-oriented network. Akamai [NSS10] is one of such those systems. It leverages several servers (more than 200k) deployed across different networks for two main goals: hosting authoritative DNS servers, to match users' requests, and locally caching delivered contents, to speed-up the delivery performance especially when multiple users are requesting the same content object. CDNs' main drawbacks are the high cost in terms of resources (bandwidth, storage, nodes distribution) and the introduction of a overhead due to the fact that the whole protocol stack is traversed during a normal transmission, which can be not negligible (for example, they usually perform multiple DNS redirections to satisfy users' requests).

Information Centric Networking, or ICN[6] [AD12] is a new approach for the conversion of current Internet into a content-oriented network. The ICN approach is purely location independent, provides content-based functionalities directly at the Network layer (such as request for a content, or send a content to a requester), and allows the development of network functionalities which were in the past either unfeasible, or implemented at the higher levels of the protocol stack. A clean-slate approach to introduce ICN (or other novel technologies) may be desirable, but it is practically unfeasible, because it would mean to replace all the network nodes in the world: the architectural side of the evolution problem then arises.

1.3.2 The architectural issue

Two main factors limit the evolvability of the Internet architecture. It first has been observed [AD11] that all layered architectures converge to a hourglass shape (cf. Figure 1.4).

[6]ICN is a generic term, including several instances. The NDN paradigm hinted in the Introduction at page 1 is one of such instances.

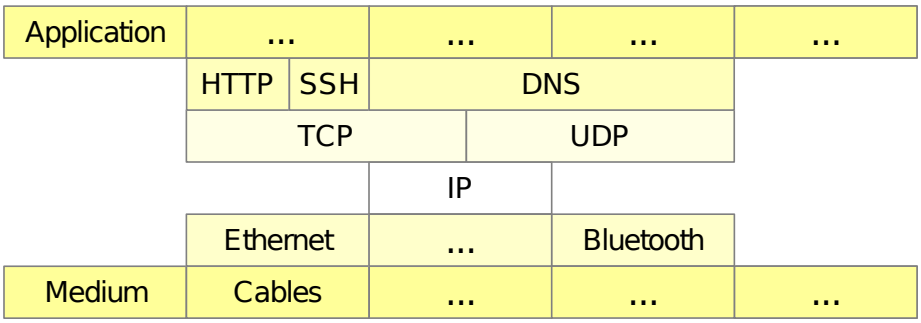


Figure 1.4: *Hourglass shape of TCP/IP protocol suite.*

Since the birth of the Internet, several applications have been developed on top of the protocol suite, while a lot of technologies have been introduced at the bottom. Therefore, evolution is very proficient on the top and at the bottom of the architecture, while in the middle there is a bottleneck. For instance, the Bluetooth and RFIDs technologies are quite recent indeed, as well as the streaming applications or cloud storage. However, all of these improvements share the same IPv4 layer proposed in the Seventies. The lack of evolution is not only caused by the layered structure of the TCP/IP. The second reason that makes the evolution of the waist very difficult is the strong coupling with the hardware. Pure hardware design is not flexible, and to introduce novel designs often requires novel equipments. We can assert that a layered architecture, together with a strong coupling with hardware, eventually leads to stagnation.

The idea of a programmable network was already known in the mid Nineties [NMN+14], and has currently recaptured the interest of the research community thanks to the proposal of **Software-defined networking** (denoted by the acronym SDN, [Ope12]). SDN's essential point is to have a software control plane connected through open APIs to a fast and reliable hardware that performs current data plane operations. The decoupling between control and data plane may facilitate architecture's innovation enabling the network data-path programmability: we assume SDN as a base to solve the Internet's architectural ossification. This enables the possibility for every actor, from network administrators to end users, to plug their own applications, services or routing protocols on top of an SDN network, drastically improving flexibility and ease of management/upgrade.

As a summary, our interpretation of the Internet evolution is twofold. On the one hand, the behavioral evolution problem can be solved by improving existing data plane using one among different enhancement proposals: for this purpose we chose to exploit the features of the Information Centric Networking paradigm. On the other hand, the architectural evolution problem is addressed by adopting a SDN paradigm to decouple the data and the control plane. In the remainder of this chapter, we present the main contribution brought in the fields of ICN and SDN.

1.4 Enhancing the Data Plane with ICN

Information Centric Networking is a communication paradigm that has recently been proposed in the research community. The main goal is to improve Network Layer to better adapt it to current Internet's usage. This is feasible when some mechanisms like forwarding and routing are based on names rather than locations. The new proposed network has some advantages, which can be summarized as: network traffic reduction, native support for advanced functionalities such as loop detection, multipath forwarding [AD12].

The ICN model inspired few research activities: among all, the *Named Data Networking* (NDN) architecture proposed by Van Jacobson [JSB+09] and Zhang [ZEB+10] is the most popular. In the following we focus on NDN, showing its architecture, the implementation, as well as its main features and the challenges that arise from such approach.

1.4.1 Architecture of NDN

In NDN, the focus is on content, including videos, images, data, but also services and resources, which are represented by some information in the network. In particular, a **content item** is any piece of information that a user may request. It is split in multiple segments called **content chunks**. Each content chunk is marked with a unique identifier called **content name**. The content name is used to request all chunks of a given content item and perform the data forwarding in the NDN network. The forwarding mechanism which takes into account a content name instead of the location of a node is called **name-based forwarding**.

There are two main naming schemes used to map content chunks to their corresponding name [Bar12]. The choice of the naming scheme may affect some properties of the NDN network, like scalability, self-certification and also forwarding simplicity.

In a **hierarchical scheme**, a content name consists of a set of components, separated by a delimiter, followed by a chunk identifier. The components can be human-readable strings that are separated by a delimiter character, similarly to what happens with web's URLs. A possible example for such a name is `<fr/inria/students/AliceBob/2>`, which represents the chunk with id equals to 2 of the content `fr/inria/students/AliceBob`. Any subset of contiguous components is called a **prefix**. This scheme allows aggregation at many levels (e.g. all content names starting with the same prefixes), and it binds different prefixes with different scopes: that is, the same local content name may be reused within a different domain (a different prefix). Conversely it makes it hard to perform operations such as a lookup on a table using the content name as key. This naming scheme is the one adopted by NDN.

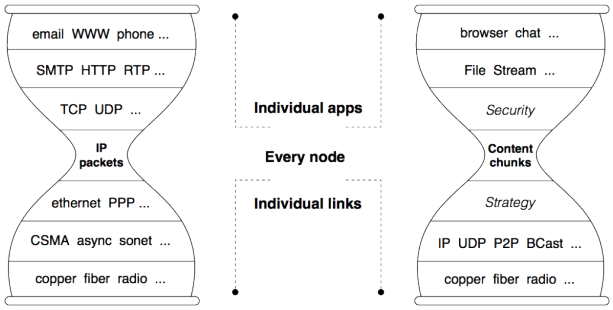


Figure 1.5: *Communication layer: similarities and differences between IP and NDN. Picture from [JSB+09].*

A **flat naming scheme** requires that a different unique content name is assigned to all content chunks, within a universal scope. A flat name is usually the hash value of some human-readable name followed by a non-hashed label. An example of a flat name [Bar12] is `<P:L>`, where P is a cryptographic hash function of a user's public key, while L is a label assigned to the content by the owner. Flat names enable easy and native self-certification functionalities and permit simple lookup mechanisms. The main drawback is the limited scalability, because aggregation is not possible due to the absence of a hierarchy.

Together with those two approaches, a less popular addressing scheme is a attribute-value pairs name scheme, in which names are similar to JSON objects.

1.4.1.1 NDN Layers

Similarly to the Internet's architecture, NDN is logically divided in multiple layers [ZEB+10], each of them responsible for a certain subset of name-based operations. Part of the effort in the NDN architecture design has been dedicated to the shift of some functionalities to the Network Layer, which is enhanced with content-awareness and some basic name-based transport mechanisms. In the following we present a bottom-up overview of the NDN Layers.

The lower layer is the *Physical layer*, which corresponds to the homonymic TCP/IP's layer. The *Transport layer* sits on top of the physical layer; at this level any transport technology can be used to carry content chunks (e.g. TCP, UDP, etc.). The transport layer is also responsible of connecting the physical layer with the above layer that is the *Strategy layer*. This layer is responsible of making forwarding decision content chunks independently of what transport technology is used. NDN's main layer is the Network Layer, similar to the IP layer, in which content chunks and content requests are assembled in packets to be transmitted. NDN enables content knowledge at the Network Layer, differently from the IP Network Layer that required to traverse the protocol stack up to the Application Layer. Moreover some functions such as congestion control or loop detection, which were in the past delegated to the above Transport

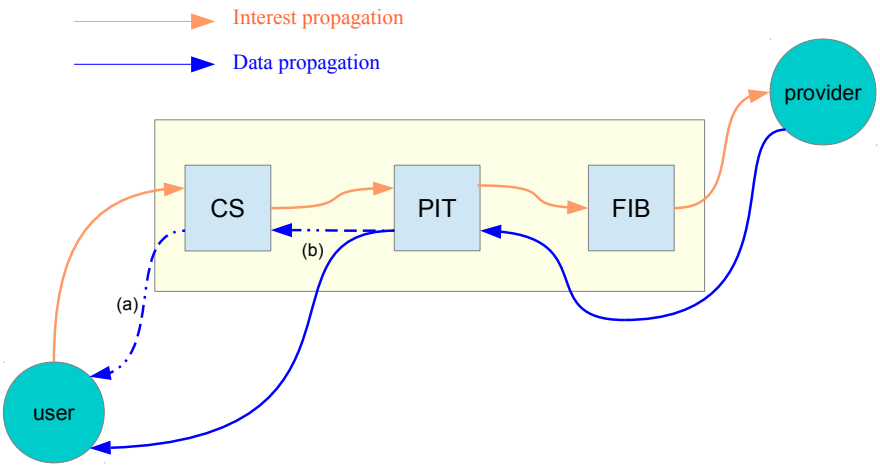


Figure 1.6: *Communication walktrough of Interest and Data packets. (a) describes when a cache hit occurs, and no further Interest propagation is needed. (b) shows that PIT multicasts Data packets towards all requesters and (optionally) sends them to the CS for caching.*

Layer, can be implemented in the NDN Network Layer. NDN proposes to have a *Security layer* just above the Network layer, in order to secure contents before reaching the application layer. Finally, on top of the stack there is the *Application layer*, in which all the actors of the communication either produce or consume contents.

1.4.1.2 Communication paradigm

The NDN network is based on two simple network functions: send the request for a content and transfer the content requested. The actors of the communication are the *Data Producer* and the *Data Consumer*. The producer is the content provider, which responds to content requests received through the network. The consumer is the user that wants to retrieve a certain content.

The communication is driven by receivers, because it starts as soon as a user wants to retrieve a content, and it finishes when the consumer has entirely received the content. This mechanism is pull-based, because the transmission begins when a content consumer requests a certain content, and the network reacts to this request and propagates it to the content producer, which eventually serves the request with the corresponding data. Even though NDN adopts a pull model, ICN does not make any assumption on the communication paradigm, and there are also push-based ICN proposals in the literature [CPW11].

The content exchange mechanism is based on two packet types: they are called **Interest packet** and **Data packet**. Figure 1.6 shows the path of user requests, which are represented by Interest packets, towards a content provider that eventually replies with the pieces of content requested

with the related Data packets. The behavior of the network is summarized in the following communication walkthrough, but it is investigated in the details in Chapter 3 and Chapter 4.

When a user[7] requests a content, it sends out a request containing the content name, which is made of a sequence of human-readable strings separated by a delimiting character, followed by a chunk identifier. The requested content name, plus the chunk identifier and some additional information are encapsulated in the Interest packet. All routers in an NDN network are capable of forwarding the Interest packet in a hop-by-hop fashion, towards the destination which is usually the content producer. We call an NDN router a **content router** (CR), and more generally we refer with the name CR to any device capable of performing content-based operations (see Section 1.4.1.3). After the request reaches the content producer, routers can send the requested data back to the source of the request. According to a certain policy, routers can also cache some contents, in order to provide it to the user without forwarding anymore the content request. Each Interest packet is usually related to a single chunk. When the request reaches the content producer it replies with the corresponding data. Similarly to the requests, every data response is encapsulated in a Data packet.

Interests are forwarded by the content routers using longest prefix matching (LPM) computed over a set of content prefixes stored in a Forwarding Information Base (FIB). Routers can forward packets over multiple interfaces, thus multipath transmission is supported. To realize symmetric routing and multicasting, NDN uses a Pending Interest Table (PIT). The PIT keeps track of a state for each packet transmission. Interests for requested content that are not yet served are saved in the PIT. A PIT's entry is the tuple `<content_name,list_interfaces, list_nonces,expiration_time>`. PIT prevents useless transmissions due to either duplicates or existing requests: when a request for the same content name is received from multiple interfaces, it simply updates the list of interfaces. The nonce is a pseudo-random number generated at the moment of the Interest creation and then binded to it. PIT detects and avoids loops of Interest packets comparing the nonce stored in the packet itself and the list of nonces stored in the PIT entry for a specific content request. To further reduce Interest transmissions, a cache called Content Store (CS) is accessed before PIT lookup. If the piece of content requested has been cached before, CS sends it back, without further access to PIT.

On the downstream path, PIT is accessed for each data packet in order to get a *breadcrumb*[8] route for that Data packet. PIT entry is therefore used to get the reverse path from the content producer to the requester in a symmetric way. Thanks to the list of interfaces saved in the PIT entry, router can send Data packets to multiple output interfaces, thus supporting native

[7] In NDN a user may be a human or a computer. If not differently specified, we use the impersonal "it" to refer to a general user

[8] The term "breadcrumb" derives from the story of "Hansel and Gretel" by Brothers Grimm, in which the children dropped pieces of bread to mark their path and eventually find their way back home.

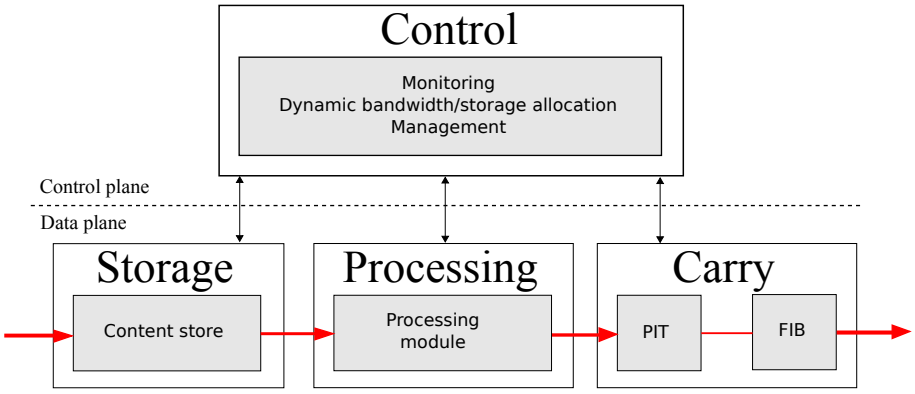


Figure 1.7: *High level block diagram of a content router.*

multicast. CS can cache Data packets, with a specified caching policy, in any of the visited routers of the downstream path from the producer to the requester(s). Caching mechanisms enhance the network efficiency because popular contents would be disseminated in the network and therefore they might be retrieved more quickly by content users.

1.4.1.3 NDN Node

An NDN Node (or content router) is a node which is capable of performing name-based operations on packets carrying name-based identifiers. It may be considered an enhanced router that forwards packets, similarly to a normal router, but also supports name-based operations.

A block diagram of a content router is shown in Figure 1.7. We can assume the usual distinction between Data plane and Control plane. We now show how each block of a content router implements the functionalities described in the NDN communication paradigm. In the Data Plane, the first element is a Carry module, which consists of a Forwarding module and a Pending Interest Table, which is responsible for Interest/Data packets propagation and symmetric routing. The *Forwarding module* is responsible of the propagation of Interest packets. It manages hierarchical variable-size content names with a potentially huge amount of state. Its main data structure is the Forwarding Information Base. The forwarding module is investigated in the Chapter 3. The *Pending Interest Table* provides a mechanism to store a temporary state for pending requests: this is achieved by storing the pending requests that still have to be served. PIT is also responsible of the backward propagation of the Data Packets. This module is designed as a data structure where all the Interests that wait to be served are stored. The PIT is investigated in the Chapter 4. A *Processing module* is responsible for any other processing that may be performed on Interest and Data Packets as popularity estimation, security verifications, etc. The *Storage module* is the Content cache, which is populated by NDN routers depending on some caching policy. The caching policy may exploit the different popularity of

the contents. The most common policies are the *Least Frequently Used* and the *Last Recently Used*. The processing and storage modules are out of the scope of this thesis.

Inside the control plane we can identify the *Control module*, which is responsible for all the control and management operations such as content prefixes distribution (i.e. routing, traffic monitoring). This thesis mainly focuses on enhancing the current data plane with NDN functionalities. A simple control plane is present, but it is not analyzed in the details.

1.4.2 Features of ICN and NDN

Since we mainly focus on NDN as main architecture, we distinguish the features of the generic ICN paradigm and the specific NDN architecture to avoid confusion between them. In fact NDN is just one of the possible realizations of ICN, and its design choices may differ from other ICN instances. The main difference between ICN and current Internet is that the content knowledge is integrated in the Network Layer. This enables several features that were not possible with the existing infrastructure or were difficult to implement.

Symmetric routing and multipath Routing protocols in IP do not grant in general any symmetry in the choice of the path between two endpoints. Conversely, the NDN lookup-and-forward process, which makes use of the FIB and PIT data structures, enables symmetric routing: Data packets always follow the reverse path taken by the corresponding Interest packets thanks to the soft state stored in PIT [ZEB+10]. Current Internet routing is asymmetric, and this is not always a desirable feature because of two main issues. The first one is described in [Pax97, ch. 8], where Paxson asserts that several protocols are developed under the assumption that the propagation time between hosts is well approximated by half of the Round Trip Time (RTT): when the paths are different in the two directions, this assumption is no longer valid. In the same way, measurements under very asymmetric networks may lead to inconsistencies when bandwidth bottlenecks have to be detected [Pax97]. Cisco [Cis] finds another problem in asymmetric networks when forwarding decisions rely on state information stored on network devices: in fact, the same traffic flow may encounter the state information required in one direction, but not in the reverse path.

NDN also natively supports multipath-routing. FIBs may store multiple next-hops for the same content prefix [WHY+12], and an ICN node can send Interest packets to different output interfaces for load balancing or service selection. Loops are prevented both for Interest and Data packets. For the Interest packets loops are avoided thanks to the PIT, because it can detect duplicate packets using content name and a random nonce. Data packets do not loop thanks to the symmetric routing feature.

Multicast If many users want to retrieve the same content, several Interest packets are produced by end nodes and propagated inside the ICN network. If multiple Interest packets (carrying the same content name) reach the same node but have a different nonce, they can be aggregated in a single PIT entry with the related interfaces from which they have been received. When a matching Data packet reaches the ICN node, it is duplicated and sent to all the interfaces stored in the corresponding PIT entry, naturally realizing multicast. Duplicate Interest packets, or Data packets that have no matches in the PIT are automatically discovered and deleted. Applications such as video conferencing or video/audio live streaming perfectly fit with both the content-based nature of NDN and the native multicast feature that it provides.

Adaptive traffic control Current Internet implements a traffic control at the Transport Layer thanks to the Transmission Control Protocol. Since TCP works at both the endpoints and not at every hop in the network, its reaction against congestion is slow and it usually follows a sawtooth-like pattern for a single TCP flow. NDN, as well as the end-to-end congestion control, may use some techniques to implement in-network congestion control, in order to reduce or augment the number of active connections [GGM12, WRN+13]. Working at the Network layer, its reaction against a congestion is faster and the flow pattern is more graceful. As an example, some internal node may detect congestion because several PIT entries expire, implying that corresponding Data packets do not match them and therefore users will not receive the content requested. The node may then lower the Interest rate (or it can decrease the PIT size): as a result, the Data packet rate decreases as well, and all the subsequent Interest packets overflowing the PIT table will be immediately dropped, which further reduce the overall bandwidth usage. When multiple links towards a destination are present, ICN nodes can also natively perform load balancing simply sending Interest packets to the multiple interfaces stored in the FIB entries.

Security and privacy NDN includes some security primitives at Network layer [ZEB+10]. First of all, content packets are signed, and every content name is mapped to a signed piece of content. Authentication and integrity are preserved with a per-content granularity: this is one of the main differences with Internet secure protocols (i.e. TLS, SSL). When Internet security mechanisms have been introduced, they mainly provided a secure channel, with a cost of reduced performance [NFL+14]. The problem of the securization in the Internet is delegated to the upper layers of the protocol stack, mainly focusing on blacklisting some untrustworthy locations, but no assumptions are made on the content transmitted through it. This could enable the paradoxical case in which fake or insecure data are transmitted over a secure channel from a trustworthy location.

Anonymity is preserved in NDN because no information about the requester is carried in the Interest packets. Data packets are signed with any kind of certification algorithm, and could possibly transmit information about the content producer. When anonymity is needed also in the producer side, signature algorithms that do not use public/private keys may be adopted.

In-network Caching Once the network has knowledge of the content flows (content-awareness) some caching mechanisms can be implemented directly at the Network Layer. In-network caching enables an easier content distribution mechanism for many reasons: popular contents, those that represent the highest bandwidth usage in the network, can be detected and cached by edge routers close to the users, providing better content retrieving performance. Moreover, when content popularity is exploited, advanced caching techniques may as well improve both storage and retrieving performance.

It is possible to find similarities between traditional caching and ICN caching, although there are many inconsistencies between the two approaches [ZLL13]. For instance, a web cache is usually located in a well-known position, whereas every ICN node can potentially cache any content, and each item can be stored at a per-chunk granularity, in contrast with the per-file granularity of regular caches: this may adverse the typical *Independent Reference Model* assumption [FGT92, ZLL13] which allows to prove the effectiveness of the classical caching algorithms[9].

1.4.3 Implementation of ICN

There are three main options for the deployment of an ICN network, which are valid for any ICN instance. The **clean-slate** approach proposes to replace IP with ICN. This option requires several changes in the existing infrastructure, and thus a long deployment time. Another approach is the **application overlay** approach, that requires to implement ICN on top of current existing network at the Application Layer. This strategy is feasible from the deployment point of view, but it will reduce some benefits such as efficiency and bandwidth saving. For this thesis we will focus on a third approach, which is called **integration** approach. The integration approach is based on the coexistence of existing network designs and ICN. In this case, a Content Router will be able to perform IP operations and ICN operations according to the type of packets it receives. This approach also intrinsically enables incremental deploy,

[9]Traditional caching algorithms have a file-based granularity, under the assumption that the probability that a given object will be requested is only determined by its popularity, independent of previous requests. NDN's caching acts at a per-chunk level: different chunks of the same content are often correlated, e.g. requests typically follow sequential order.

because network administrators can incrementally upgrade their devices without changing all their routers.

1.4.4 Challenges

There are several challenges and open issues that emerge both from the development of a ICN network and from the integration of ICN functionalities inside the current Internet Layer. This section provides a detailed description of the main challenges, focusing on those that are resolved thanks to this work and giving an overview of the others.

1.4.4.1 ICN forwarding

One of the major differences between a content router and a regular IP router is the large amount of state [PV11]. The number of domains registered under a top-level domain may be of tens of millions[10], and the number of Google's indexed pages is even greater, allegedly reaching tens of billions[11]: to serve all web pages at least implies that the address space of ICN is larger than the 32-bit IPv4 address space; moreover, even though the IPv6 addressing scheme uses 128-bit addresses, it still has a known fixed size, while ICN naming is potentially unbounded and variable-sized components are allowed.

A large state implies that FIBs may store several elements (name prefixes in NDN) compared to IP forwarding tables. Authors of [SNO13] claim that the possible growth of the state would cause the number of element stored in a content router's FIB to be as great as many millions, while a core IP router has to manage about 500k thousand entries. Besides, IP and NDN FIBs differ also in the typology of the entries. In a regular IP router, a FIB usually contains a single next-hop information [rg16] and the possible alternatives, together with additional information, are stored in a separate Routing Information Base (RIB). FIBs are created by the main route controller starting from the RIB, and distributed to all the line cards. In NDN a FIB contains multiple next-hops, to natively implement multipath forwarding, and statistics management or other processing.

With ICN not only the FIB size increases, but the size of the stored elements does likewise. A content prefix, similarly to what happens with URLs, may consist of several components of different size. The length of each component and of the whole prefix is not predictable, and no regularity is required. It entails that existing hardware optimizations for fixed-size 32-bit IP addresses cannot be easily converted and patched for an NDN content router.

[10] `http://research.domaintools.com/statistics/tld-counts/`
[11] `http://www.worldwidewebsize.com/`

Finally, a regular IP router must sustain a forwarding rate of tens of Gigabits per second, or even hundreds if it is part of the core network. A content router ready to be deployed must perform the lookup process, together with the packet I/O, at similar speed.

These challenges are addressed in Chapter 3, where we mostly focus on name-based forwarding on a content router which is capable of performing name lookups at a rate of tens of Gbps.

1.4.4.2 State management

The ICN paradigm requires that a soft state of every request has to be kept for a period of time. As stated in Section 1.4.1, this is done by the Pending Interest Table.

In contrast with a FIB, the PIT is accessed for every packet received from the content router (both Interest and Data packets). The PIT operations (i.e. insert, update, or delete) are performed at a per-packet granularity, being a time-critical operation in an ICN content router.

The PIT must be able to detect whether an incoming Interest packet must be added to the table, or if it can be aggregated with an existing packet previously stored, or if it is to be dropped (duplicate Interest, or other problems). Then, it must perform the corresponding action to the packet. When Data Packet reaches the PIT, it must perform a lookup-and-delete process to find whether a matching Interest has been seen, and duplicate the Data packet for every interface stored in the PIT entry.

Furthermore the PIT may reach a size that can be in the order of 10^6 entries, as the authors in [DLCW12] describe. Performing line-rate operations with a large amount of state is one of the major challenges for the deployment of ICN.

In Chapter 4 we propose a design and an implementation of a PIT that can work at high-speed, addressing these major challenges.

1.4.4.3 Other challenges

We now describe other major challenges that arise from the ICN approach.

Routing The distribution of the routing instructions, that is the development of ICN routing protocols, is still an interesting challenge for the ICN deployment. When the routing information is represented by content prefixes rather than IP prefixes the major difficulty is scaling with such an enormous state. Some papers [HAA+13] and technical reports [WHY+12, AYW+]

propose to develop ICN routing protocols that leverage the equivalent of OSPF design principles, or extending the IPv4 Map-n-encap scalability solution. Anyway, these results leave still some open issues like the naming dynamicity.

Caching Per-file caching mechanisms implemented for Web-pages [BCF+99] and P2P services [SH06] show a popularity pattern that follows the Zipf (resp. Mandelbrot-Zipf) model. Analytical models for ICN cache networks [MCG11, ZLL13] as well as experimental evaluation testbeds [MSB+15, RRGL14] are still at an early stage and reflect a research direction. Caching decision policy (e.g. which elements are stored in the current node) and caching replacement (e.g. which elements are evicted from the cache) are major challenges in the ICN community.

The deployment of such a network of caches can present some decision challenges: quoting Pavlou's keynote speech [Pav13], they can be summarized in "cache placement" (where to put caches), "content placement" (which content is allowed in caches) and "request-to-cache routing" (how to redirect requests to a cache in the proximity).

Fine-grained security Every Data Packet in NDN is signed to meet the purpose of integrity and origin verification [ZEB+10]. Signature and verification algorithms may introduce a not negligible overhead, and then limitate network performance. Confidentiality is granted by encryption algorithms for every chunk of content: scalability is then a major issue.

Network attacks The immunity to DDoS-like attacks is another ICN challenge. Old issues such as cache poisoning [TD99] and new ones, i.e. Interest Flooding attacks [AMM+13] open new research directions.

1.5 The SDN approach for an evolvable Network

As described in Section 1.4, the hourglass shape of the current Internet's architectures allowed evolution on top and bottom of the protocol stack, the network layer being the bottleneck. In addition, the Internet is designed with a client/server paradigm in mind, which does not fit current traffic patterns.

One of the main difficulty in the evolution of networks is related to the tight coupling between hardware devices and their functionalities. When network administrators want to add new functionalities to their network, most of the time they are forced to add new hardware, sometimes

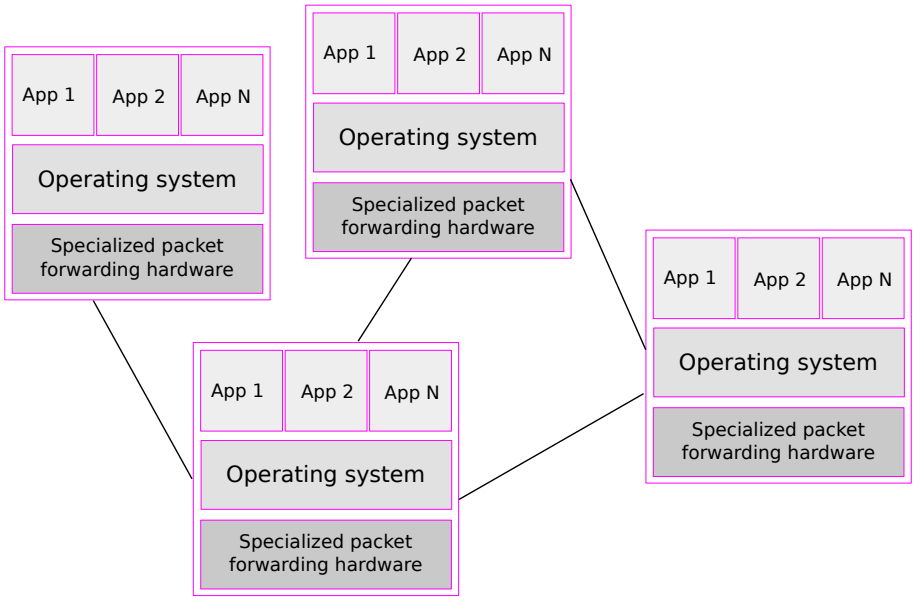


Figure 1.8: *A closed Internet architecture.*

even from the same vendor, with a consequent high cost in terms of time, management, monetary investments. Software-defined networking (SDN) proposes to decouple the data plane, which is hardware-based, and the control plane, which can be implemented in software.

In the following sections, we first describe the SDN architecture, then we focus on the implementation, its advantages and the main challenges.

1.5.1 Architecture

Current Internet architecture is shown in Figure 1.8. The network is made of different network devices, each of them with its own operating system and a specialized hardware dedicated to the packets forwarding and route computations. In other words, Data and Control planes are tightly coupled. This design is limiting for the requests of a market that is evolvable and demanding. We now highlight the motivation behind the new architectural needs [Ope12].

Motivation We may group the issues behind the Internet architectural problem in four main categories: rise of new traffic patterns, complexity, scalability and dynamicity.

In Section 1.4 we considered ICN as a new network paradigm that exploits current infrastructure. This is not the only emerging paradigm: cloud computing is already a standard for enterprises, and data centers are more and more common. Internet access from mobile devices is also bringing mobility as a requirement for novel networks.

Nowadays, network developers tend to meet these new needs by introducing some ad-hoc solutions, usually on the top of the protocol stack. This one-protocol-per-problem strategy can be observed at the Application layer in the Internet protocol stack. This leads to the problem of complexity.

When networks consist of several devices implementing different sets of instructions (routers, firewalls, servers) management is not an easy task. *Vendor dependency* is one of the main issues, because operating systems, management functions and in general any configuration function may be different for different device producers. Also, as previously mentioned, new functionalities require new hardware, and even replacing a single device may require several configuration steps, firewall settings, and any protocol-dependent action that may cause misconfiguration or deployment delays.

As a consequence of complexity, the scalability property of current infrastructure is reduced. In details, scaling issues have two main reasons. The first one is the difficulty of management described in the previous paragraph, which implies several configuration steps as soon as a new hardware is deployed to meet the market's requirements. Resources overprovisioning may be a solution to reduce the number of upgrades. Network dimensioning should be performed to estimate the amount of bandwidth, computational power, and any other kind of resource needed by a specific application, but this is not possible when the traffic pattern is unknown. The second problem arises from the fact that today's requirements are almost unpredictable: previous works [SDQR10, ABDDP13] showed that the performance of cloud application of a major vendor such as Amazon are affected by high variance and traffic unpredictability: overprovisioning is no more a feasible solution to scalability because of the difficulty in estimating both the modifications affecting the current traffic pattern and the new resources' requirements.

The scalability issue is the cause of the difficulty of the network to evolve in order to meet new demands. On the one hand, the conjunction of all the aforementioned problems causes the network to be slow-reacting to any innovation. On the other hand, users demand may vary with a timescale that is no more in the order of years, as it was in the past, but is now reaching a smaller window (months or even less). For instance, mobile applications are developed and distributed in the market on a daily base. The equivalent can be said for computer software. Networks are not able to be as dynamic as their corresponding software counterparts.

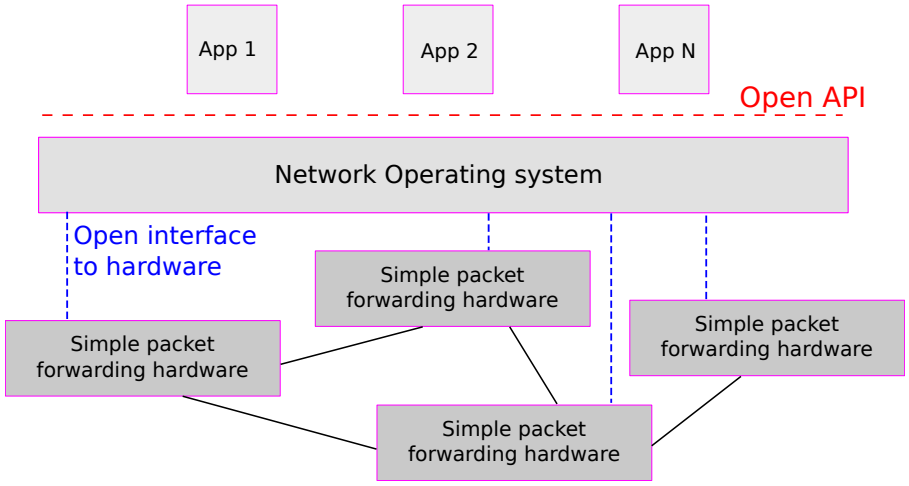


Figure 1.9: *The SDN proposal of an evolvable network.*

The SDN approach SDN proposes to solve these problems by decoupling Control and Data Plane. Figure 1.9 shows the network elements of an SDN network: all network nodes are replaced by **simple packet forwarding engines** that are vendor-independent, mostly implementing hardware functionalities. They share a common interface with some open APIs towards an open **Software Control Layer**, which is programmable via software, enabling a higher level of customization both for users and network administrators. Hence, all the hardware-related operations can be limited to the simple action of forwarding packets, while the overall behavior of the network is decided by a logically centralized network controller. Therefore, the whole network becomes a programmable infrastructure.

Decoupling data and control plane To achieve the goal of a software programmable network, SDN proposes to separate the control plane from the data plane. This is shown in Figure 1.10. The SDN architecture has a lower *Infrastructure layer*, which consists of all the simple forwarding devices. Since those devices have the goal of transmitting packets, they belong to the data plane. No management instructions are performed at this level, and all the control operations are delegated to the above layer which is the *Control layer*. SDN control softwares contained in the control layer manage the behavior of the underlying infrastructure, as well as the network services provided to the top-tier level. An open interface resides between the Data plane and the Control plane: all the network devices and the network services communicate using this interface. Finally, there is the *Business layer*, in which all the user applications are deployed. Such those applications exploit the APIs shown by the Control layer to change the network behavior according to the needs.

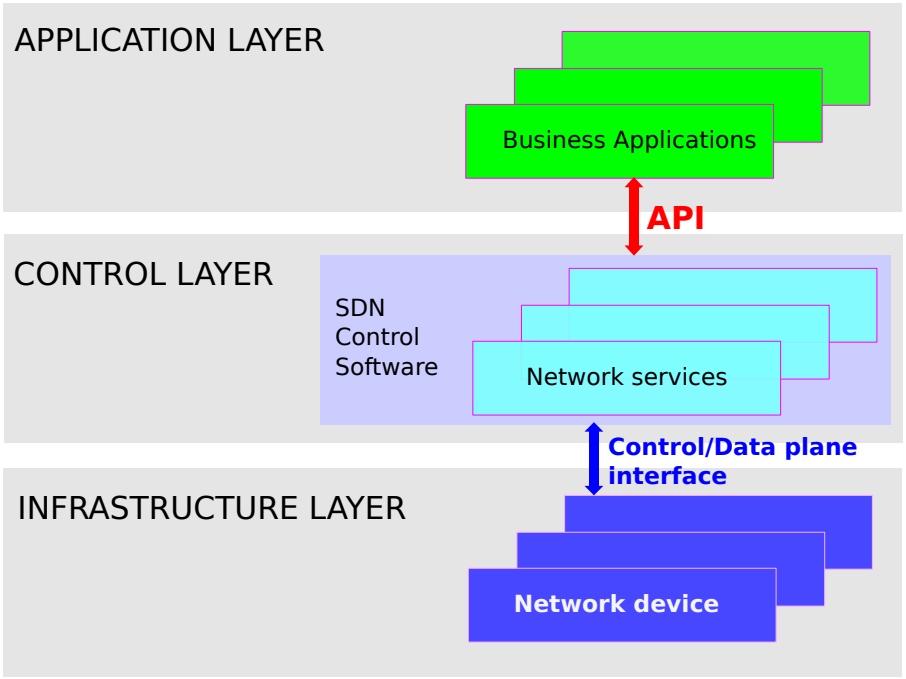


Figure 1.10: *SDN architecture.*

1.5.2 Implementation of SDN

Figure 1.11 shows the impact of the deployment of SDN on existing architecture. The left side of the figure represents a typical existing router: the control plane, represented by the route engine, and the data plane, which consists of several line cards, coexist in the same device. SDN proposes to split Data and Control plane functions, creating the three main entities of the SDN architecture:

- A centralized SDN controller, in which software network programs are deployed;
- A set of distributed SDN forwarding engines, which operates according to the policies specified by the network controller;
- An open protocol to allow the communication between the data and the control plane: the OpenFlow is an example of such a communication protocol.

SDN controller The SDN controller is a computer program that may execute one or more SDN instances. Each instance implements a particular behavior of the underlying network. The controller communicates with the network devices by sending and receiving control messages in a separate control channel. The controller instructs forwarding engines on how to process

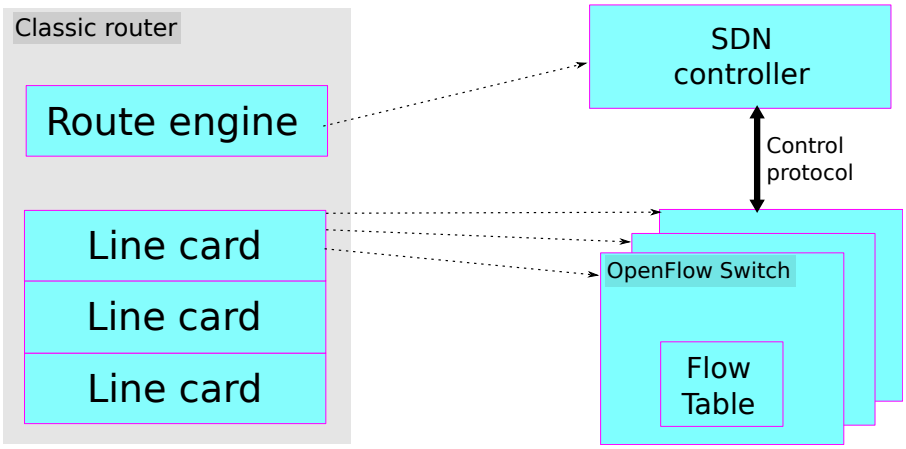


Figure 1.11: *A block diagram of a traditional router, and the corresponding SDN elements for each block.*

packets by installing some **network rules**. Every rule specifies a set of fields in existing IP packets that should be matched against incoming packets, and the action to perform on matching packets. The controller usually reacts to forwarding engine queries (see below), but it can also proactively install rules.

Forwarding engines SDN-enabled routers are simple network devices that are able to forward packets according to the rules stored in a (set of) Rule table(s). Rule tables are populated by the controller. When a packet is received, it looks up the rule table. If no entry is found, then the router sends the packet to the controller, that can decide the type of the action that will be performed for that packet (and all the following packets of the same flow). The controller can therefore decide to create a path between two or more nodes, and send OpenFlow rules to all the devices in the path. In this way, all subsequent packets of the same flow are directly forwarded by the Data Plane without involving the controller.

Control protocol The standard control protocol for SDN networks is OpenFlow [MAB+08]. OpenFlow is a protocol defining control messages that are exchanged in an SDN network (usually between the controller and the forwarding engines), and the format of the rules that are installed in the SDN-enabled routers. A rule is usually a set of fields in an IP packet (for instance, IP source and destination, MAC source and destination), but it may also define some bits at arbitrary positions in the IP header using wildcard masks[12]. A rule also defines an action that can be *forward*, if the packet is forwarded to a next hop; *drop*, if the packet should

[12]Current Internet aggregates routes sharing the same prefixes with the Classless Inter-Domain Routing approach [FLYV93]: when two or more destinations share the same next-hop, and their addresses are contiguous, current node can store just one entry with a prefix which includes the destination addresses, to save memory and reduce the update cost. The prefix matching feature can be easily implemented in SDN by simply enabling matching only for the field of IP destination, using IP prefixes as rule predicates.

be dropped; *custom* if a customized action must be performed before reaching a forwarding decision such as sending packet to the controller or performing a lookup in another rule table.

1.5.3 Features of SDN

Abstraction and modularity In the field of the Computer Science, abstraction and modularity are commonly used.
The term **abstraction** indicates the definition of a certain software component specifying only the interface showed to the users. Programmers can therefore make their own design and implementation choices, as long as they respect the external interface. As an example, we can consider the abstract data type of a "List", which is a set of elements that allows insert and remove operations. The abstract model of the List describes its behavior, that is storing elements without particular ordering and providing "add" and "remove" mechanisms. The List can be implemented in several ways (using arrays, linked nodes, or even with a dictionary), but all those implementations are compatible as they share the same user interface.

Moreover, when programs consists of many millions lines of codes, they are built using **modular programming**. Rather than building a huge monolithic application, developers today tend to build modular software, usually with a main software core and multiple ad-hoc modules. Such a software is easy to be upgraded, because the core is stable and new modules may be added when needed. Software evolution is much easier when modularity is preserved.

SDN brings the notions of abstraction and modularity to the network. Programmers can implement their own network instance, and it can be made of several modules, fully upgradeable on the fly. Flexibility is one of the main advantages of those features.

Virtualization Network virtualization is the coexistence of multiple virtual network instances sharing the same physical infrastructure. In other words virtual networks are a set of virtual nodes and virtual links, mutually independent and logically separated, built on top of real nodes and real links [CB10]. This approach is similar to the homologous equivalent in servers: several computer instances rely on the same physical architecture, but they are logically separated and independent. Many technologies are currently used to implement virtual networks, and they are located at different levels of the protocol stack: a Virtual Private Network (VPN) is a virtualization technique implemented on top of IP/MPLS networks, and therefore it belongs to the layer 3 [RR06]; a Virtual LAN (VLAN) consists of a set of switches that may virtualize physical links using a special field in the layer 2 Ethernet frame [TFF+13]; there are even layer 1 virtualization techniques, using L1 protocols such as SONET or SDH [TIAC04].

SDN plays a major role in improving existing virtualization mechanisms. There are two main advantages in SDN-based network virtualization [JP13]. The first one is the ease of deployment: thanks to the simplified data plane and the centralized software programmability, it is easy and fast to install several network instances at different locations of the network without requiring ad-hoc hardware or protocols. Finally, as SDN switches are simple standard devices while the intelligence resides in the centralized controller, they are expected to be cheaper than current devices where the control is distributed on each equipment.

Network Functions Virtualization, or NFV [C+12, HGJL15], is a further decoupling of network functions from the dedicated hardware devices (similarly to Virtual Machines in virtualized servers). While SDN virtualization mainly focus on network resources, NFV aims to abstract the functions implemented by network devices (e.g. firewalls, packet filters, DNS servers) and relocate them as generic boxes to be deployed anywhere in the network infrastructure. NFV may work in conjunction with SDN by providing the abstract infrastructure (Data plane) that is orchestrated by the SDN controller (Control plane). Furthermore, a generic SDN/NFV network may be thought as a set of virtual network instances where the abstract virtual functions may deployed as simple network virtual devices, all sharing the same physical infrastructure.

Interoperability When the network behavior is software-defined, the definition of network instances becomes similar to programming a computer, providing a gain in terms of flexibility and ease of management. Furthermore, when the APIs between the Software and the Hardware sections are open and publicly available, every vendor may decide to adopt such a model, giving a great improvement in the interoperability. Network owners are not restricted to use vendor-compatible hardware, and vendors might also acquire some market slices thanks to the increased competition available.

1.5.4 Challenges

The emerging SDN model opens several research challenges. In this section, we provide a detailed description of the challenges that we address in this thesis, and overview the remaining open issues.

1.5.4.1 Network modeling and problem detection

Network administrators are interested in diagnosing problems in their networks such as the existence of loops, or black-holes. This task can be performed by analyzing control plane configurations or checking the data plane forwarding tables.

Network verification based on analysis of network topology and the nodes' forwarding tables is more promising: in fact, it is hard to generalize complex protocols and configuration languages used for the management of control plane configurations; on the contrary, data plane analysis of forwarding tables can detect bugs that are invisible at the level of configuration files [MKA+11] and can quickly react to network changes [KZZ+13, ZZY+14].

Nevertheless, the table verification problem is a complex task, especially in systems where forwarding rules are specified over multiple fields covering an increasing number of protocol headers as in SDN. In particular, verifying network problems by checking forwarding rules is classified under the NP-Hard complexity category (cf. Section 5.1). Some tools implement verification algorithms exploiting heuristics, or other properties which are tightly coupled with the Internet protocols. Two issues emerge from these approaches: they are not easy to extend for a network which is far different from the current Internet network, and the implementations are very specific for the network problem(s) they aim to solve.

We developed a mathematical framework, inspired from (but not limited to) SDN, which can be used to model any kind of existing networks. We are able to use our framework to analyze router's forwarding behavior and detect a provable complexity bound in checking the validity of a network through forwarding tables validation. Our framework, covered in Chapter 6, makes use of an implicit representation of the header classes: therefore we called it *IHC*[13].

1.5.4.2 Other challenges

We provide an overview of other challenges that are not addressed by this thesis.

Scalability/feasibility SDN forwarding engines should be able to perform fast lookup in their rule tables. Those tables must be able to reach a throughput of tens of Gigabits per second for an edge-network. Rule tables should be able to rapidly modify the entries in case of rule modification, refresh or other management activities. The SDN controller can limit the size of the network it can manage due to the bandwidth and CPU processing requirements. In [YTG13] authors express their concern about the feasibility of an SDN-controlled network when several updates per seconds have to be performed on many routers (e.g. a data center is expected to require more than 30k updates/s).

[13]We mostly use the name IHC to identify the software tool implementing our framework. This is an on-going work, and perspectives are given in the Conclusion at page 143

Control traffic management The SDN architecture assures a centralized network controller. This may cause a control traffic bottleneck in the proximity of the SDN controller, especially when the network is made of several nodes. Moreover, the disruption of a link to the controller may cause the entire network fault, if some backup solutions are not provided. Even though SDN requires a logical centralization, SDN may use several controllers, deployed in a distributed fashion [DHM+13]. This opens some challenges about the associated coordination algorithms, scalability and reliability especially under adversing traffic conditions (e.g. several Control Plane updates, that may affect the coherence between the controllers' state).

Part I

Data Plane Enhancement

Chapter 2

Introduction to the First Part

Since the birth of the Internet, network measurements observed a highly changing behavior of the network usage, with many evolving patterns of traffic and sensible increment in the traffic volume. The introduction of new services and the technology advancements are among the main catalysts of such an evolution process. Several challenges had been faced to satisfy the upcoming requirements of the evolving Internet. Some of them still have to be addressed: when network dynamics change frequently, to develop new network systems accordingly is not always an easy task. We observed (cf. Background, Section 1.3, page 12) that today most of the users are interested in retrieving some content from the network, without caring about the exact location of a specific content producer. The Internet is moving in the direction of consuming content, information and services independently from the servers where these are located: it makes sense to imagine that most of the network traffic is being replaced by content requests and data. In fact, recent experiments conducted over a two-year timescale showed that nowadays' Internet inter-domain traffic belongs mostly to large content/hosting providers. Moreover, the greatest part of such a traffic migrated to content-related protocols and applications, such as HTTP, video streaming and online services [LIJM+11].

The new approach in using the Internet comes together with a relevant increment in the amount of traffic. The Internet's traffic can be thought as a set of different components: it is useful to observe that the global growth is not uniform, and the traffic change may affect only a subset of components. CISCO's measurements on wired and wireless Internet [Ind14, Ind15] show a fine-grained traffic type characterization of the Internet's traffic. Table 2.1 displays a projection of the network traffic, based on a combination of analyst forecasts and direct data collection of the last years, showing that the trend may dramatically increase both for fixed and mobile networks: the overall Internet traffic is expected to grow in the next years.

Network Type	Traffic volume per year (PB/month)						CAGR
	2014	2015	2016	2017	2018	2019	
Fixed	31,545	37,908	46,511	58,115	72,933	91,048	24
Mobile	2,050	3,430	5,599	8,906	13,587	20,544	59

Table 2.1: *Projection of the annual growth rate of both Mobile and Fixed Internet traffic, in the period 2014-2019, and the corresponding compound annual growth rate (CAGR). [Ind14]*

Traffic Type	Traffic volume per year (PB per month)						CAGR
	2015	2016	2017	2018	2019	2020	
Internet video	21,624	27,466	36,456	49,068	66,179	89,319	33
Web, email, and data(*)	5,853	7,694	9,476	11,707	14,002	16,092	22
P2P file sharing	6,090	6,146	6,130	6,168	6,231	6,038	0
Online gaming	27	33	48	78	109	143	40

Table 2.2: *Projection of the increment in the overall Internet traffic per traffic typology, in the period 2015-2020, and the corresponding compound annual growth rate (CAGR) [Ind15].*
(*) data includes generic unclassified data traffic excluding file sharing.

Table 2.2 shows as well the overall traffic increment classified per typology: on-demand video show the highest compound growth rate[1], followed by general HTTP and data application. Online gaming is a novel traffic typology, becoming more and more popular, as shown by its CAGR, while classical file sharing can be considered to stay almost constant.

The Named-data Networking (NDN) paradigm is a recent networking vein that proposes to enrich the network layer with name-based functionalities, which are novel communication primitives centered around content identifiers rather than their location. The key idea behind this approach is to identify these new communication primitives in order to implement them directly at the network layer. We may macroscopically classify two content-distribution primitives: requesting for a content, and sending back the requested content.

While the evolving network usage may be matched by the new networking paradigms arising in the research community, the traffic growth has to be matched by a corresponding increase in network devices' performance: in order to sustain a higher traffic volume, routers have to show better performance in terms of forwarding speed and memory consumption. In a study of 2000 [Rob00], authors already foresaw the Internet's growth trend, and they claimed that "to keep pace with the Internet's growth, the maximum speed of core routers and switches must increase at the same rate." High speed network design is a critical topic for current and future R&D.

[1]The compound annual growth rate (CAGR) is the mean annual growth rate of a value, calculated over a specified period of time longer than one year.

Both traffic typology and network paradigm are changing, and moreover the volume of the network traffic becomes higher and higher. The question that may arise is: how does the network architecture react to this change?

The integration of content distribution functionalities in network equipments is of critical importance for the deployment of future content delivery infrastructures. On the one hand, today's content delivery infrastructures are evolving towards in-network solutions where content distribution is more and more integrated with the underlying ISP networks. On the other hand, the NDN paradigm recently proposed by the research community, proposes to provide content distribution functionalities as native network primitives.
However, the integration of content distribution functionalities in high speed routers imposes severe changes to today's hardware and software technologies. For instance: the routable address space is expected to be several orders of magnitude larger than today's IP, requiring new routing and forwarding algorithms to handle it; packet-level caching requires the design of storage engines that should operate at high speed, and can be coupled with forwarding mechanisms; soft-state schemes can be required to enable symmetric routing; content traffic monitoring and optimization tools should be integrated in network equipments in order to dynamically adapt routing, forwarding, and storage management strategies.

Despite several name-based strategies have been proposed, few have attempted to build a content router. Classical solutions (e.g. the CCNx application [Cen]), are typically implemented as userspace (or kernelspace) daemons, and work at application level using commodity PCs and commercial-off-the-shelf network cards, which are not suited for large scale deployment and high-speed networking. Our work fills such gap by designing and prototyping Caesar, a content router for high-speed content-distribution. We build Caesar as a small scale router that is totally compatible with current hardware and today's protocols. Caesar's design was inspired by two main research directions: First, the need of integrating advanced content-aware functionalities in current network devices without changing all the infrastructure, but rather improving the existing Data plane by adding name-based features. Second, Caesar must work in conjunction with existing high-speed routers, and therefore it must sustain a throughput per link of several gigabits per second (Gbps).

The goal of this part of this Ph.D. thesis is to focus on system design of a new networking architecture based on named-data and on its performance evaluation by means of experiments. Hereafter we try to provide one among the possible answers to the demand for advanced content-based functionalities in high-speed devices. This introduction serves to set up the theoretical and practical foundation of this part of the thesis. We investigate the design principles in Section 2.1, and explore the design space in Section 2.2. Section 2.3 introduces the hardware that we used to develop our system. Section 2.4 describes both the methodology and the

testbed that we used for the implementation and the experiments. Finally, Section 2.5 shows our main contributions and the organization for the remainder of this part. From now on, several symbols are introduced: we provide a table of symbols at page 103.

2.1 Design principles

We focus on the design, the implementation and the evaluation of a flexible high-speed content router. For our router design we assume the typical separation between *data plane* and *control plane* (cf. Background Section 1.2.2, page 11). The data plane consists of $\mathcal{N}$ line cards which operate at a rate R. The line cards, which are logically separated in input and output line cards, are interconnected by one or more switch fabric with an overall rate of $\mathcal{N}R$. We associate to each line card an identification number LC_i, which is a progressive integer number assigned to the i-th line card. The control plane consists of a route controller, which calculates route updates received from neighbor routers, and compute the best next hops. Since the control plane is out of the scope of this work, we assume a simple control plane with pre-calculated routes. *Caesar* is the name we give to our content router.

To achieve the design of an evolvable and scalable system we keep in mind three major design directions:

D.1 allow content-based networking, to simplify the content-distribution over current Internet;

D.2 perform high-speed packet processing, to target the increment in network throughput;

D.3 be compatible with current technology (i.e. forwarding of regular both IPv4 packets and content requests/data), to avoid a clean-slate approach.

The direction **D.1** requires to develop name-based algorithms that can handle the large name-space imposed by a content-oriented architecture and can be implemented in high-speed network equipments. Our schemes should be flexible to work on different environments (e.g. core, edge, intra/inter-AS), and can further be optimized to better exploit the different content replicas available in other network equipments, and to influence items availability across the network.

The point **D.2** impacts the choice of router chassis as well as the selection of the type and number of line cards used. We must consider a trade-off between scalability and performance: for example, software solutions are often easy to deploy and integrate, but this comes at the cost of losing in performance; on the contrary, hardware solutions may perform better in a very

specific environment, but they are usually difficult to implement in real environments in the short term.

Finally, the point **D.3** is about the coexistence with existing infrastructure. Flexibility is fundamental for an extensible network architecture. A clean-slate approach is not always possible when several nodes are present in a network and those nodes are not designed to be furtherly improved. An integration (and evolvable) approach allows to gradually deploy novel functionalities, without affecting the behavior of the existing architecture and older devices.

2.2 Design space

We consider in this section the main directions for the design space exploration of Caesar. We start discussing the prerequisites required to our data structures. Then we analyze the algorithms requirements needed to access the chosen data structures. Finally we explain the approach of data structure placement.

Data structures Data structures are of fundamental importance in the design of a content router. Differently from a regular IP, an NDN-enabled data structure may contain any human-readable delimiter-separated string. The amount of state is greater than current IP (cf. Background Section 1.4.4, page 23): both the size of the table (number of elements) and the size of a single element may potentially be unbounded. The data structures used in an NDN high-speed router have to be designed to support fast lookup (at line rate), and optimization are required to store a large number of elements while minimizing memory footprint. Since access speed is often in contrast with small memory requirements, a trade-off must be taken into account.

The full design of a content router includes many data structures (cf. Background Section 1.4, page 15). We consider a Forwarding Information Base (FIB), a Pending Interest Table (PIT) and a Content Store (CS). The FIB, similarly to a current IP forwarding information base, contains a set of matching rules and one (or multiple) next-hop information. The PIT is a data-structure used to store pending requests not served yet. The CS is a packet-level cache used to temporary store forwarded data to serve future requests.

Algorithms Choosing a computable set of steps to access a desired element stored in the data structure is part of the algorithm design. Conventional algorithms have an execution time that usually depends on the input size. In NDN the input is a content name, which may be

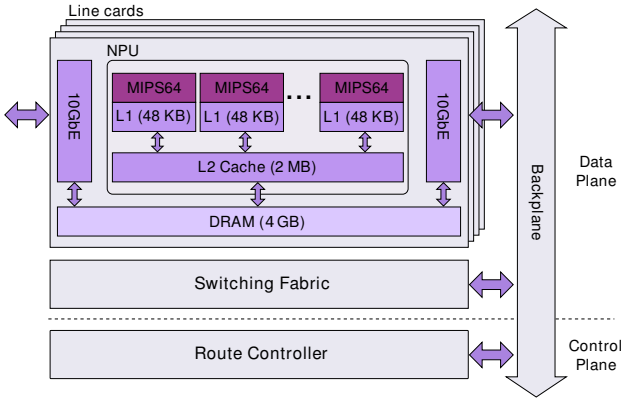


Figure 2.1: *The hardware architecture of Caesar's Forwarding Engine.*

made of several variable-size components. The main goal for the design of a good algorithm is a deterministic execution time, as much independent as possible from component length and number of components.

We can access an element of a data structure by means of exact matching or prefix matching. An exact-match algorithm consists of finding the elements whose bits are exactly the same as the content name carried in the input packet. We talk about longest-prefix matching when we are interested in finding the string that shares the greatest number of components with the name carried in the input packet.

Placement The data structure are usually located in a specific line card. The placement consists in deciding *where* a data structure should be located in a router. Some functionalities may be enabled or disabled when different placements are used.

This part of the design space mainly refers to the optimization of the placement of data structures inside a router in order to support all functionalities. Some classic examples of placements are input-only, output-only and input-output. As the name suggests, this affects the positioning of some data structure inside the input or the output line card, or even in both of them.

2.3 Architecture

The hardware description of our system has been previously introduced in [VP12]. The design of our content router targets a network device for an enterprise network, i.e. few 10 Gbps ports and cumulative speed that depends on the number of active slots.

We chose to use a Cavium Network Processor equipped with CN5650 line cards (LC). Figure 2.1 shows the hardware architecture of our content router. The chassis mainly consists of a

micro telecommunications computing architecture (μTCA) containing twelve slots for advanced mezzanine cards (AMCs). Every slot may contain one single line card, and all the line cards are respectively connected among each other by means of an internal switch. The internal connection with the switch is called *backplane*. Each line card is equipped with a network processor unit with 12 cores at 800 MHz with 48KB of L1 cache per core, 2MB of shared L2 cache memory, a 4-GB off-chip DRAM, a SFP+ 10GbE interface[2] , and a 10 Gbps interface to the backplane. We make use of a different number of slots of the chassis, depending on the desired maximum throughput. When the Cavium unit is under extreme conditions, i.e. traffic coming from all the line cards, the internal switch can sustain such a rate introducing some latency, which however is negligible.

Our content router is modular: each additional functionality can be thought as a module equipped in our content router. Caesar's design targets a small-scale router that is easily deployable in current networks, e.g. via a simple firmware upgrade of existing networking devices. This constraints the hardware choice to programmable components already widely adopted by commercial network equipments. We therefore resort to network processors optimized for packet processing. The Data plane is designed to be backward compatible with existing networking protocols. In particular, its switch fabric is based on regular L2 switching, and thus name-based forwarding is implemented on top of existing networking protocols (e.g. Ethernet and IP) in a transparent fashion, without requiring a clean slate approach.

2.4 Methodology and testbed

The design and implementation of our solutions eventually resulted in a small-scale prototype: one of the contributions of this thesis is the experimental evaluation of such a system. We performed several experiments, following the guidelines provided by the RFC 2544 [BM99], which inspired some of our tests. We describe the methodology of the performance evaluation and the experiment categories in Section 2.4.1, then we introduce our testbed in Section 2.4.2. Finally, Section 2.4.3 describes the properties of our workload.

2.4.1 Methodology

A general experiment works as following: the device under test (DUT) is connected by means of commodity fiber wires to some test equipment, in a bidirectional way. The test equipment

[2]The acronym SFP stands for *Small factor pluggable* and identifies a set of tranceivers for connecting optical fibers at rate of 10 Gbits per second with existing L2/L3 network interfaces (see `http://www.cisco.com/c/en/us/products/collateral/interfaces-modules/transceiver-modules/data_sheet_c78-455693.html`)

sends out packets at a tunable rate to the DUT, which performs some processing depending on the functionality to be evaluated, and sends them back to the tester. In order to be a valid test, each transmission should last at least 60 seconds. Two main types of experiments are used in this thesis: throughput and latency computation.

Throughput It is the fastest rate at which the DUT can process packets received by the test equipment without any loss. Since the DUT has finite computational capacity, given that the maximum transmission rate of the LCs is fixed (10 Gbps), it may be unable to sustain such a rate for all packet sizes. To give two examples, transmitting packets of size 1500 bytes at the rate of 10 Gbps requires the DUT to process about $834k$ packets every second, while a size of 200 bytes at the same rate translates into about $6.25M$ packets processed every second.

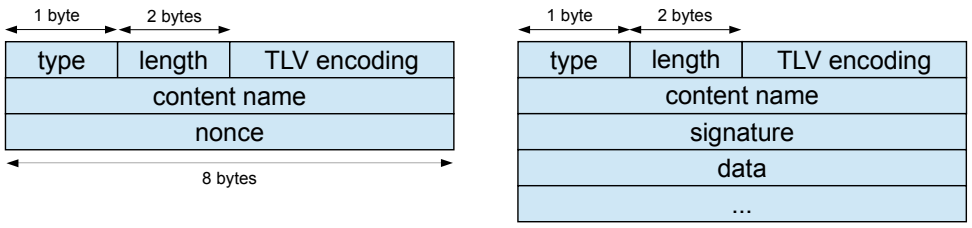
Therefore, throughput measurements typically consist in finding the smallest packet size at which the DUT is not affected by any packet loss, and its value is measured in millions packets per second. The results of the throughput test may be reported in the form of a plot, with the x coordinate being the packet size and the y coordinate being the calculated packet rate. The same plot may be reported choosing as x coordinate any other variable that may affect the throughput.

Latency Once the throughput is calculated (and therefore no packets are lost at the throughput rate) it is possible to calculate the latency as the difference between the time a packet is fully transmitted by the tester, and the time at which the corresponding packet is received, processed and completely sent back to the test equipment. The results of the latency test are reported as a table showing the average, maximum and minimum calculated latency.

2.4.2 Test equipment

Our testbed consists of a commercial traffic generator, Caesar 's chassis (the device under test), and a set of network interfaces.

Traffic generator Our traffic generator is equipped with 10 Gbps optical interfaces, connected through optical fibers to Caesar's line cards. The traffic generator is tuned to inject 10Gbps traffic of Interest and Data packets. In order to generate NDN packets, the traffic generator is extended to produce regular IP packets with our additional name-based header. Two line cards of the traffic generator are used to generate Interest and Data packets respectively.



(a) *Interest packet format.* **(b)** *Data packet format.*

Figure 2.2: *Header format of Interest and Data packets. This header is appended to regular IPv4 packets, at the level of the UDP protocol.*

In the basic configuration, these line cards are connected to two separate line cards of Caesar, but the number of line cards involved may vary depending on the experiment to be performed.

Device under test Caesar is activated with a set of line cards plugged in the chassis. The router's line cards may work in full-duplex mode; however, it is more convenient to separate the transmission and reception processes in order to fully understand the behavior of the system: for this reasons, our line cards are configured to work in half-duplex mode and act either as input or as output line card exclusively.

Packet headers A standard header format for ICN is currently under debate in the ICN research group at the IRTF [icn]. In absence of such standard, we use our own header which consists of four fields. First, the 8-bit *type* field, which is a value that identifies if current packet is an Interest or a Data packet. Then a 16-bit *length* field specifies the size of the *content name* field. To expedite parsing, we also include a modified TLV[3] *components* field, that consists of an 8-bit number of components in the content name, and several *offset* fields, each containing an 8-bit offset for each component in the content name. For backward compatibility, the name-based header is placed after the IP header, which allows network devices to operate with their standard forwarding policy, e.g. L2 or L3 forwarding. After the variable-size TLV and content name field, the Interest packet contains the 64-bit *nonce* field, used to detect looping Interests (cf. Background Section 1.4.1.2, page 17). The Data packet does not have a nonce field, but contains the signature of the content provider and the actual data to be consumed. Our Interest and Data packet format is summarized in Figure 2.2a and 2.2b.

[3]TLV, acronym of Type-Length-Value, is a flexible way of storing some optional data. It consist in a first field *Type*, which identifies the information category; it is followed by the *Length* field, containing the size of the data to be parsed, and finally the actual data, stored in the *Value* field.

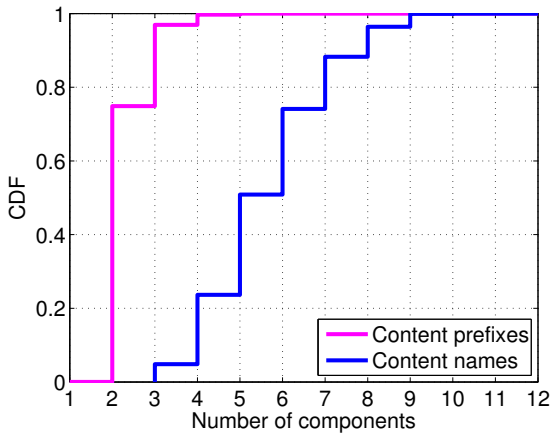


Figure 2.3: *Reference workload. Distribution of number of components.*

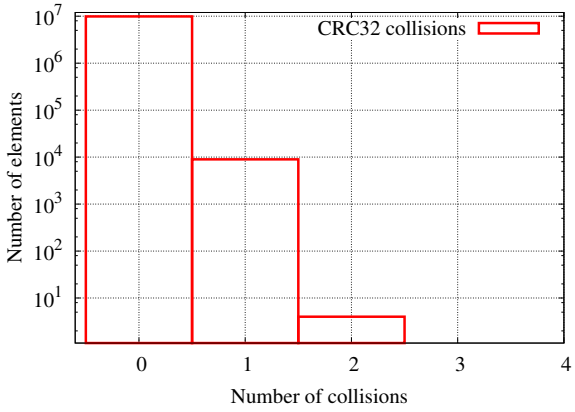


Figure 2.4: *Number of prefixes with colliding hash values. Our reference hash function is from [KC04]. The collision rate of the workload is* $p_c = 0.002$.

2.4.3 Workload

We call *workload* the combination of a set of content prefixes representing the items to be stored in a table and the related requests. As a *reference workload*, we use the trace presented in [WZZ+13]; this trace contains $n = 10^7$ content prefixes the authors collected by crawling the Web. As done in [WZZ+13], we generate the requested content names by adding random suffixes to content prefixes randomly selected from the trace; also, we form, on average, 42-Bytes long names as in the reference workload. We denote the average name length with ν, i.e. $\nu = 42$ for our trace.

The prefixes of the reference workload are common URLs: this fits well with the hierarchical naming scheme of NDN (cf. Background Section 1.4.1, page 15). Each component of a prefix is therefore the string separated by the delimiter `"/"`. Throughout the evaluation, we also use several *synthetic workloads* to understand the impact of workload parameters and adversarial traffic patterns.

Properties of the reference workload Figure 2.3 shows that most of the content prefixes in the reference workload are composed by less than 3 components (2 components on average), while content names extracted from the content requests have a number of components between 3 and 12 components (4 components on average).

When some elements are stored in a data structure, each of them is usually placed in a bucket[4] together with the hashed value of the item itself. Our underlying architecture may exploit fast

[4]The term *bucket* refers to a fast-access location in a data structure, such as the index of an array. Hash functions are used for the index computation.

hash values computation whose performance depend on the quality of the hash typology: e.g. cryptographic hash functions need more computational resources than generic hash functions, but result in a better distribution of the hash values). We make use of a classical CRC32 [KC04] algorithm for the computation of hash values (cf. Section 3.3.3).

The effectiveness of the CRC32 may be analyzed considering the number of items of the workload hashed to the same hash values. This is resumed in Figure 2.4, which plots the number of hash collisions our reference workload. The plot shows that the majority of the elements have no collisions, i.e. all these prefixes have a unique CRC32 value. Less than ten thousands prefixes have one collision, and only 6 elements have two hash collisions. We calculate the probability of collision p_c as the ratio between the items colliding at least once and the size of the dataset. In our reference workload we have $p_c = 0.002$.

2.5 Contributions

Chapter 3 is related to the design and implementation of Caesar's Forwarding module. We show that it can handle a throughput of 10 Gbps and a forwarding table containing over 10 million elements (some orders of magnitude greater than current IP forwarding tables). In addition, GPU offload further speeds up the forwarding rate by an order of magnitude, while distributed forwarding augments the amount of content prefixes served linearly with the number of line cards, with a small penalty in terms of packet processing latency. This work has been published in [PVL+14b].

In Chapter 4 we focus on the PIT module of Caesar. We designed and implemented a data structure that can support fast updates (both insert, remove, update instructions) at line-rate. In addition to the data structure, the design process requires to identify the PIT placement as well, which refers to where in a router the PIT should be implemented. Similarly to the Forwarding module, we provide numerical and experimental results on the PIT module of Caesar. Previous work has been published in [VPL13].

Chapter 3

Forwarding module

This chapter presents the design, implementation and evaluation of an NDN-enabled forwarding module. An important data structure is accessed in the forwarding module: the Forwarding Information Base (FIB).The forwarding module is accessed in the Data-Plane for the Interest Packet propagation. In the NDN communication model (cf. Background Section 1.4.1.2, page 17) every Interest packet should be sent towards the provider of the content requested by the user, until the content or a copy of it is eventually found. Since a lookup for every packet is required, this gives emphasis to the high-speed capabilities that our module should support.

The Chapter is organized as follows: we introduce the motivations in Section 3.1, which also describes the goals and the features of our forwarding module. We investigate the design space in Section 3.2, and then we present the design and the implementation of our high-speed NDN-enabled forwarding module in Section 3.3. We extensively evaluate our module in Section 3.4, using our prototype coupled with a commercial traffic generator and both synthetic and real traces for content prefixes and requests. Section 3.5 concludes this chapter with a summary of the results.

The main findings of this Chapter are that our forwarding module can sustain 10 Gbps links assuming packet size of 188 Bytes and FIB with up to 10 million prefixes. We also show that some extensions can be added to our prototype in order to support both a larger FIB and higher forwarding speed.

3.1 Description

A forwarding module is a module capable of performing I/O of incoming packets, and forward them to a destination chosen after performing a lookup in a Forwarding Information Base. The Forwarding Information Base is a data structure that stores a set of entries: classical FIB entries contain a matching rule and a corresponding next-hop destination. In the case of an IP router, the rules are represented by IPv4 prefixes, and next-hops are output interfaces. In NDN, the FIB rules represent name prefixes with one (or more) next hop(s) information. We adopt a DNS-like naming scheme proposed by NDN [JSB+09], where content items are split in a sequence of packets identified by a content address, a hierarchical human-readable name with $(d+1)$ components delimited by a character: d components form the *content name*, whereas the last component identifies a specific packet, for example `/fr/inria/thesis.pdf/packet1`, where the delimiter is `"/"`. A content router maintains forwarding information for *content prefixes* that are formed by any subset of components from the content names, for example `/fr/inria/*`. FIB is accessed in two cases: first, the control plane can modify the table inserting new entries to populate the next hops information, or updating/deleting existing entries due to some routing change; second, the data plane retrieves the information for the actual packet forwarding.

When an Interest packet arrives at a line card, the forwarding engine identifies the interface where the packet should be forwarded performing a *longest prefix matching* (LPM) algorithm on the FIB using the name carried by the packet. Additional processing such as policy routing and packet classification might be performed depending on the router features. Afterwards, the packet is moved to the line card where the output interface resides via the backplane and the switch fabric. I/O operations are eventually performed in order to forward the current packet towards its next hop. FIB is not accessed during the propagation of Data packets: thanks to the NDN feature of symmetric routing, Data packets transmission is delegated to the PIT module (see Chapter 4).

The forwarding module may be considered the most complex module of a content router's architecture [PV11], both in terms of the functionalities it enables, and for the challenges it presents. We identify two major challenges. First, our design choice of an integration approach implies that our content router has to process at least the same amount of packets processed by a today's router, assuming forwarding tables that are several orders of magnitude larger. Second, the NDN-like forwarding tables are populated with string prefixes that may consist of several components and (possibly) unlimited characters per component.

We propose to perform a Longest Prefix Matching algorithm on content names, whose main novelty is the *prefix Bloom filter*, a Bloom filter variant that exploits the hierarchical nature of content prefixes.

The forwarding module is the first step to build a content router, for which we chose the name *Caesar*. Caesar's forwarding module replicates the design of a a classic router where each line card stores a full copy of the FIB. As in classic routers, each line card implements our algorithm for name-based LPM. A switch fabric then moves packets from input to output line card upon LPM decision. We then extend Caesar's forwarding module by proposing *distributed forwarding*, a mechanism which allows to share the FIB across Caesar's line cards to maximize the overall FIB size. A second extension is *GPU offloading*, where name-based LPM is partially delegated to a Graphics Processing Unit (GPU), with the goal to further increase the rate.

3.2 Design space

In this section we explore the design space for the FIB module design. Section 3.2.1 reviews the related work. Section 3.2.2 justifies the choice of a two-stage algorithm to optimize lookup speed. Then, in Section 3.2.3 we analyze a set of *data structures* which could be used for the FIB design. Our design is then analyzed in the details in Section 3.3.

3.2.1 Related work

We describe the proposed solutions addressing the same challenges of this chapter, namely data structures for name-based LPM and NDN forwarding scheme design.

Name-based LPM Related work on name-based LPM may use three types of data structures: hash table, Bloom filter and trie solutions.

Cisco [SNO13, SNOS13] implement LPM using successive lookups in a hash table. Rather than using the common longest-first strategy (lookups start from the longest prefix and continue with shorter prefixes until a match is found), the search starts from the prefix length where most FIB prefixes are centered, and restarts at a larger or shorter length, if needed. This approach bounds the worst case number of lookups, but cannot provide average performance bounds.

Authors of [DLWL15] propose a Bloom-filter aided hash table, where the FIB entries are first stored in a counting Bloom filter [FCAB98], a variant of regular Bloom filters (cf. Section 3.2.3) used to detect a bucket that is less charged, and finally stored in a regular hash table. To accelerate lookup, up to k auxiliary counting Bloom filters are used, where k is the number of hash functions applied on the incoming content name. Though this solution provides good results in terms of lookup speed, its main drawbacks are the memory consumption (several

data structures are used) and the poor insert/delete performance, due to the multiple accesses to the main memory.

NameFilter [WPM+13] is an alternative name-based LPM algorithm employing one Bloom filter per prefix length. At lookup, a d-component content name requires d lookups in the different Bloom filters. This approach has two intrinsic limitations. First, it cannot handle false positives generated by the Bloom filters, and thus packets can eventually be forwarded to the wrong interface. Second, it cannot support additional functionalities, such as multipath and dynamic forwarding.

Among the trie-based solutions, Wang *et al.* [WHD+12] propose the name component encoding (NCE), a scheme that encodes the components of a content name as symbols and organize them as a trie. Due to its goal of compacting the FIB, NCE requires several extra data structures that add significant complexity to the lookup process, and result in several memory accesses to find the longest prefix match.

Differently with these approaches, we target a deterministic number of memory accesses when detecting the maximum prefix length. We also aim to detect false positives, and supports enhanced forwarding functionalities.

NDN router To our knowledge, the only known attempt to build a content router, i.e. a router capable of performing name-based forwarding, are [SNO13] and [YC15].

In [SNO13, SNOS13] a content router is implemented on a Xeon-based Integrated Service Module. Packet I/O is handled by regular line cards, while name-based LPM is performed on a separate service module connected to the line cards via a switch fabric. Real experiments show that the service module sustains a maximum forwarding rate of 4.5 million packets per second. Simulations without packet I/O show that the the proposed name-based LPM algorithm handles up to 6.3 Mpps.

Authors of [YC15] propose to use the hardware engine acceleration provided by Intel, called DPDK [Int], to perform NDN I/O on regular network card, and a LPM algorithm based on binary search with multiple prefix hash tables, one for each prefix length. Given d as the maximum number of components of prefixes, d hash tables are created: these tables form a balanced binary search tree. Lookups occur with a binary search of the prefix hash tables to detect in which table the prefix match can be found. With this solution they are able to store up to 10^9 prefixes, providing good lookup performance when multithreading is used. The drawback of their approach is the significant performance loss when performing insert/delete on the prefix tables; moreover, their experiment are conducted with short names (only 2 bytes) which are not realistic for a real deployment.

Different from [SNO13, SNOS13], we want to support name-based LPM directly on I/O line cards in order to reduce latency, increase the overall router throughput, and enable ICN functionalities without requiring extra service modules. Our purpose in the design of Caesar is to achieve performance that are still comparable to [SNO13, SNOS13] even though our prototype is based on a cheaper technology.We also face the work in [YC15] since our dataset use realistic traces with content names of 42 bytes. Additionally, we address the problem of FIB growth allowing LCs to share the content of their FIB in order to support greater tables.

3.2.2 Algorithm

We recall, as described in Section 3.1, that at the reception of an Interest packet the forwarding module performs a LPM on the incoming content name, selects the next-hop information and sends the packet towards the chosen output LC. Since the processing required to detect the prefix with the longest match with the incoming content name is easily affected by its number of components (that we remark being variable-sized and potentially unbounded), we design our algorithm to be as much deterministic as possible.

We propose a two-stage algorithm that is able to detect at (almost) constant time the longest prefix stored in our FIB. The stages of the algorithm are:

1. detect the length of the longest prefix matching the incoming content name;
2. continue with regular hash table lookup processing with the longest matching prefix.

The first step may be achieved by means of a data structure for membership query [Blo70, BM04]. This kind of data structures provide an efficient representation of a set of any type of element, and can be used to test whether a specific element is represented in the set or not. In particular, the feature that we exploit is that any variable-size element is compressed with a fixed-sized length. The membership query does not provide the effective element stored in a corresponding data structure: this task is delegated to the second part of the algorithm, which accesses a different data structure.

We show in Section 3.3 that we manage to amortize both the prefix length and the number of its components thanks to our two-stage approach.

3.2.3 Data structure

We leverage two categories of data structures for the two stages of our algorithm: Bloom filters and hash tables.

The Bloom filter naturally fits with the first step of our two-stage algorithm: it is a data structure that provides a simple membership query function. Bloom filters convert any variable-size string to a tunable number of bits. k bits are chosen with k independent hash functions applied to the whole string, and are then set in a bit string of size M according to the pre-computed hash values. The membership query consists in checking the value of the k bits calculated using an incoming string as input: if one or more bits are "0", then the element is certainly not in the set, while if all bits have value equal to "1", then the element may be present with a false-positive probability. The probability of error may be tuned, and depends on the number of hash functions, the size of the filter and the number of stored elements n.

We derive a closed formula representing the probability of error under the assumption of uniform hashing, which implies that hash functions may point to any bit location with the same probability. We start considering the first insertion and the first hash function: under these hypothesis, the probability of setting a bit to one is $\frac{1}{M}$. Consequently, the probability that a certain bit is still zero after the first insertion is $(1 - \frac{1}{M})$. We remind that for a single element insertion we set k bits in the Bloom filter using k hash functions: if all the k hash functions are mutually independent, we may consider any bit set as a result of an independent experiment. Therefore, the probability of finding a zero after k bits are set is $(1 - \frac{1}{M})^k$. The same analysis holds for n elements, considering every single insertion as an independent experiment when the n items are mutually independent. After the Bloom filter is populated with n elements, the probability of still finding a bit set to zero is $(1 - \frac{1}{M})^{kn}$, and the probability of finding a bit set to one is $1 - ((1 - \frac{1}{M})^{kn})$. The probability of error is equivalent to the probability of receiving a positive answer to the membership query while the element is not present in the set. This translates in performing k bit tests, by means of the k hash functions calculated over the incoming item, finding all bits set to 1. We can approximate the probability of error with the formula $p_{error} = \left(1 - \left(1 - \frac{1}{M}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/M}\right)^k$.

To store name prefixes we use a hash table. When a multi-core architecture is used (cf. Section 2.3) many cores may access, process and consume the same entries, causing concurrency issues. We may classify the hash tables in *lockless* and *locked* hash tables. When a locked hash table is used, the concurrent accesses are managed using locking mechanism such as mutex and semaphores: these are control mechanisms, which are usually natively built-in depending on the hardware architecture. They provide atomic access to the same memory location (e.g. one-core-at-a-time). On the other hand, when a lockless data structure is used, no control

mechanisms are provided and all cores may possibly access the same memory area. To avoid race conditions (e.g. concurrent modifications of the same memory area which cause system crashes) some other control is therefore required, like avoid shared memory and making each core access only a specific area during write operations.

Thanks to the difference of operation rate between the control and the data plane, we assume FIB as a lockless data structure: insert operations are not frequent, and occur only when some changes are detected by the Control Plane. For this reason, there is no need to lock the memory area to grant a one-at-a-time access during the forwarding operations. We then chose a lockless shared hash table, with two advantages: no control mechanisms are required, and parallel access may be exploited to augment throughput. The design of our hash table is described in Section 3.3.4.

3.3 Forwarding module: design and implementation

In this section, we detail our two-stage name-based LPM algorithm used in the forwarding module of Caesar. Section 3.3.1 and 3.3.2 describe the prefix Bloom filter (PBF) and the concept of block expansion, respectively. Both are used in the first stage to find the length of the longest prefix match. Section 3.3.3 shows our technique to reduce the number of hash values computation. Section 3.3.4 describes the hash table used in the second stage, and the optimizations introduced to speed up the lookups.

3.3.1 Prefix Bloom Filter

The first stage of our name-based LPM algorithm consists in finding the greatest length of a prefix matching an incoming Content name. To perform this lookup, we introduce a novel data structure called *prefix Bloom filter* (PBF). The PBF exploits the hierarchical semantics of content prefixes to find the longest prefix match using (most of the times) a single memory access.

Similarly to the Bloom Filter, PBF is a space-efficient data structure which is made of several blocks. Each block is a small Bloom filter: its size is chosen to fit one or multiple CPU cache lines of the memory in which it is stored (e.g. DRAM caches, not to be confused with CPU caches, are usually 64-128 MBytes, while CPU caches are typically smaller than 2 MB).

During the population of the FIB, a content prefix is stored both in the FIB and into a block

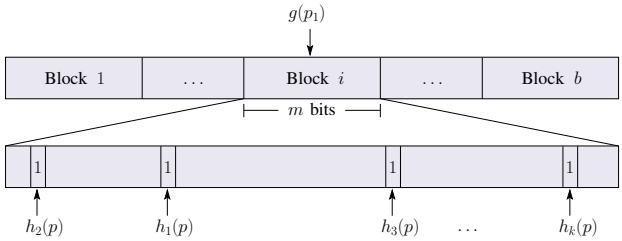


Figure 3.1: *Insertion of a prefix p into a PBF with b blocks, m bits per block, and k hash functions. The function $g(p_1)$ selects a block using the subprefix p_1 with the first component, and bits $h_1(p), h_2(p), \ldots, h_k(p)$ are set to 1.*

chosen from the hash value of its first component[1]. During the lookup of a content name, its first component identifies the unique block that must be loaded from memory to cache before conducting the membership query. The possible load unbalancing issues due to that choice are investigated in Section 3.3.2.

A PBF comprises b blocks, m bits per block, with k hash functions. The overall size of the PBF is $M = b \times m$. A scheme of the insertion of a content prefix is shown in Figure 3.1. Let $p = /c_1/c_2/\ldots/c_d$ be a d-component prefix to be inserted into the PBF. The hash value resulting from a hash function (chosen on purpose to be as much uniform as possible) $g(\cdot)$ with output in the range $\{0, 1, \ldots, b-1\}$ determines the block where p should be inserted. Let now $p_i = /c_1/c_2/\ldots/c_i$ be a subprefix with the first i components of p, such that $p_1 = /c_1$, $p_2 = /c_1/c_2$, $p_3 = /c_1/c_2/c_3$, and so on. We choose a block in the PBF identified by the value $g(p_1)$, which is the result of the hash computation performed from the subprefix p_1 defined by the first component. This implies that all prefixes starting with the same component are stored in the same block, which enables fast lookups (the possible load balancing issues are discussed in Section 3.3.2). Once the block is selected, the values $h_1(p), h_2(p), \ldots, h_k(p)$ are computed using the entire prefix p, resulting in k indexes within the range $\{0, 1, \ldots, m-1\}$. Finally, the bits of positions $h_1(p), h_2(p), \ldots, h_k(p)$ in the selected block are set to 1.

To find the length of the longest prefix in the PBF that matches a content name $x = /c_1/c_2/\ldots/c_d$, the first step is computing the hash value corresponding to the first component of the content name, $g(x_1)$. This gives the index of the block where x or its subprefixes may be stored. The block is then loaded and a match is first tried using the full name x, i.e. maximum length. The bits of the positions $h_1(x), h_2(x), \ldots, h_k(x)$ are then checked and, if all bits are set to 1, a match is found. Otherwise, if some of the bits are set to zero, or if a false positive is detected (cf. Section 3.3.4), the lookup continues trying the prefix x_{d-1}. These trials continue until a match is found or until all subprefixes of x have been tested. At each membership query, the bits of the positions $h_1(x_i), h_2(x_i), \ldots, h_k(x_i)$ are checked, for $1 \leq i \leq d$, accounting for a maximum of $k \times d$ bit checks per name lookup in the worst case. Checking the bits of a block requires

[1]In our naming scheme the first component is the equivalent of the top-level domain name in the DNS hierarchy, such as `com/` or `fr/`. For more details about DNS, refer to [Pos94].

a single memory access as the block is stored in one cache line: no further memory accesses are required for every subprefix lookup. The insert and lookup procedures are represented in pseudo-code in Algorithm 3.1 and 3.2.

Algorithm 3.1: Insert procedure for a PBF.

```
Procedure insert(components[d], blocks[b], hashfunction[k])
    /* First, we identify the block using the g() function on the first
       component of the incoming prefix                                  */
    hash := g(components[0]) ;
    block := blocks[hash  MOD  b];
    /* Then, we set k bits in the chosen block, using k hash functions on
       the whole prefix                                                  */
    For each h ∈ hashfunction[k] do
        set_bit_in_block(h(components[0...p]), block);
```

Algorithm 3.2: Lookup procedure for a PBF.

```
Procedure lookup(components[d], blocks[b], hashfunction[k])
    /* Select the block using the g() function on the first component of
       the incoming name                                                 */
    hash := g(components[0]) ;
    block := blocks[hash  MOD  b];
    /* Then, we do the lookup in the chosen block                        */
    For each i ∈ range(d, 1) do
        /* Check all subprefixes, starting with the longest              */
        For each h ∈ hashfunction[k] do
            check_bit_in_block(h(components[0...i]), block);
        If miss then
            continue at i = d − 1;
        else
            match;
```

The false positive rate of the PBF is computed as follows. If n_i is the number of prefixes inserted into the i-th block, then the false positive rate of this block is $f_i = (1 - e^{-kn_i/m})^k$. We consider two possible cases of false positives. First, assume the worst-case scenario where the name to be looked up and all of its subprefixes are *not* in the forwarding information base. In this case, assuming a content name with d components, d lookup trials are required until it can

be considered as not in the FIB, and therefore the number F_i of false positives in the i-th block follows the binomial distribution $F_i \sim B(d, f_i)$. For the Boole's inequality, the probability of a collection of events is no greater than the sum of the probability of each single event. Using a closed formula, we have that, given a collection of events A_i, then $\mathcal{P}(\cup_i A_i) \leq \sum_i \mathcal{P}(A_i)$. This inequality, called the *union bound*, allows us to calculate an upper bound for the false positive probability of a single block for a prefix lookup. After the first lookup, a sub-prefix p_{d-1} is tested, and so on up to p_1. Every sub-prefix is an event, and the collection of such events share the same false positive probability f_i, being located in the same PBF block i. We can bound the false positive probability of the PBF, thanks to the Boole's inequality, with the value $d \times f_i$, that is d trials with false positive f_i. The union bound is always valid since it represents an upper bound; it results in a good approximation for the false positive when $(d \times f_i) \ll 1$.

We now calculate the false positive probability for the PBF considering all blocks. Let $\mathcal{P}(b_i)$ be the probability that the first component of the content prefix is stored in the i-th block. When the function $g(\cdot)$ is uniform, and therefore the first components of every prefix are uniformly distributed over the PBF blocks, each block is chosen with probability $\mathcal{P}(b_i) = 1/b$ for all i, where b is the number of blocks, and thus the average false positive probability in the PBF for a content name with d components and no matches is $\frac{d}{b}\sum_{i=1}^{b} f_i$, where $f = (1/b)\sum_{i=1}^{b} f_i$ represents the average false positive rate and d is the number of lookup trials.

We consider now the case where either the content name or one of its subprefixes *are* in the table, and let l be the lenght of its longest prefix match. In this case, a false positive can only occur for a subprefix whose length is larger than l, i.e., the l-component subprefix is a real positive and the search stops. We assume the same hypothesis of the previous scenario, that is the union bound for every lookup trial and the uniform distribution of the first components. The number F_i of false positives in the i-th block then follows the binomial distribution $F_i \sim B(d - l, f_i)$, and following the same approach as the previous scenario, the average false positive probability in this block is $(d - l) \times f_i$. In general, for a d-component name whose longest prefix match has length l and under the assumption of the uniformity of the $g(\cdot)$ function, the average false positive in the PBF is well approximated by $(d - l) \times f$.

3.3.2 Block Expansion

Since prefixes starting with the same root (that is the first component) are stored in the same block, the single memory access allows fast lookups at the cost of a reduced accuracy of the PBF. In fact, when many prefixes in the FIB may share the same first component, then the corresponding block may yield a high false positive rate. This is usually avoided when the first component distribution is uniform. To avoid an increase in the false positive probability even

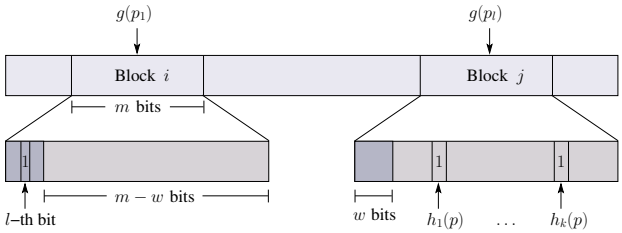


Figure 3.2: *Insertion of a d-component prefix p into a PBF using block expansion. If block $i = g(p_1)$ reached its insertion threshold or if the l-th bit is set in its bitmap and $l \leq d$, then p is inserted into block $j = g(p_l)$.*

with non uniform distributions of the first components, we propose a technique called *block expansion*. It consists in redirecting some content prefixes to other blocks, allowing the false positive rate to be reduced in exchange for loading a few additional cache lines from memory[2].

Block expansion is used when the number n_i of prefixes in the i-th block exceeds the threshold $TV_i = -(m/k)\log(1 - \sqrt[k]{f_i})$ selected to guarantee a maximum false positive rate f_i. For now, assume that prefixes are inserted in order from shorter to longer lengths[3]. Let n_{ij} be the number of j-component prefixes stored in the i-th block. If at a given length l the number $\sum_{j \leq l} n_{ij}$ exceeds the threshold TV_i, then a block expansion occurs. In this case, each prefix p with length l or higher is redirected to another block chosen from the hash value $g(p_l)$ of its first l components. To keep track of the expansions, blocks keep a bitmap with w bits. The l-th bit of the bitmap is set to 1 to notify that an expansion at length l occurred in the block. If the new block indicated by $g(p_l)$ already has an expansion at a length e, with $e > l$, then any prefix p with length e or higher is redirected again to another block indicated by $g(p_e)$, and so on.

Figure 3.2 shows the insertion of a prefix $p = /c_1/c_2/\ldots/c_u$ in a PBF using block expansion. First, block $i = g(p_1)$ is identified as the target for p. Assuming that the threshold TV_i is reached at prefix length l, block i is expanded and the l-th bit of its bitmap is set. Since $l \leq u$, a second block $j = g(p_l)$ is then be computed from the first l components of p and, assuming block j is not expanded, positions $h_1(p), h_2(p), \ldots, h_k(p)$ of this block are set to 1.

The lookup process works as follows. Let x be the prefix to be looked up, and $i = g(x_1)$ be the block where x or its LPM should be. First, the expansion bitmap of block i is checked. If the first bit set in the bitmap is at position l and x has l or more components, then block $j = g(x_l)$ is also loaded from memory. Assuming that no bits are set in the bitmap of j, prefixes x_l and higher are checked in block j. In case there are no matches, then prefixes x_{l-1} and lower are checked in block i.

[2]Block expansion requires the control plane to completely recalculate the prefix Bloom filter and distribute it to the corresponding line card. This operation is performed offline, after a threshold of prefixes per block is reached. However, it could be possible to realize an online expansion mechanism, but this is out of the scope of this thesis.

[3]The dynamic case, where the content prefixes in the FIB change over time, is addressed by the router control plane, and is explained in Section 3.3.6.2.

Algorithm 3.3 shows the pseudo-code implementation of the lookup in a PBF with expansion.

Algorithm 3.3: Lookup procedure for a PBF with expansion.

```
Procedure lookup(components[d], blocks[b], hashfunction[k])
    /* Select the starting block using the g() function on the first
       component of the incoming name                                  */
    hash := g(components[0]) ;
    block := blocks[hash  MOD  b];
    /* Check the bitmask to identify expanded blocks                   */
    expanded_blocks[] := check_bitmask(block);
    /* Then, we do the lookup in the expanded blocks (at most d different
       blocks are loaded)                                              */
    For each i ∈ range(d, 1) do
        /* Check all subprefixes, starting with the longest            */
        For each h ∈ hashfunction[k] do
            check_bit_in_block(h(components[0...i]), expanded_blocks[i]);
        If miss then
            continue at i = d − 1;
        else
            match;
```

The false positive rate of the PBF with block expansion is similar to the case without expansion, except for two key differences. First, the block size is now $m - w$ bits, since the first w bits of the block are used for the expansion bitmap. The range of the hash functions h_i is thus $\{0, 1, \ldots, m - w - 1\}$. Second, the number n_i of prefixes inserted in each block i is now computed from the original insertions, minus the prefixes redirected to other blocks, plus the prefixes coming from the expansion of other blocks.

3.3.3 Reducing the number of hashing operations

Hashing is a fundamental operation in our name-based LPM algorithm. For the lookup of a given content prefix $p = /c_1/c_2/\ldots/c_d$ with d components and k hash functions in the PBF, a total of $k \times d$ hash values must be generated for LPM in the worst case, i.e. the running time is $O(k \times d)$. Longer content names thus have a higher overall impact on the system throughput than shorter names. To reduce this overhead, we propose a linear $O(k+d)$ running time hashing scheme that only generates $k + d - 1$ seed hash values, while the other $(k - 1) \times (d - 1)$ values

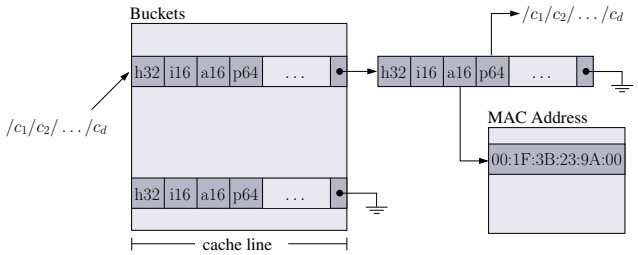


Figure 3.3: *The structure of the hash table used to store the FIB. Each bucket has a fixed size of one cache line, with overflows managed by chaining. Each entry consists of a tuple $\langle h, i, a, p \rangle$ that stores the next hop information.*

are computed from XOR operations.

The hash values are computed as follows. Let H_{ij} be the i-th hash value computed for the prefix $p_j = /c_1/c_2/\ldots/c_j$ containing the first j components of p. Then, the $k \times d$ values are computed on demand as

$$H_{ij} = \begin{cases} h_i(p_j) & \text{if } i = 1 \text{ or } j = 1 \\ H_{i1} \oplus H_{1j} & \text{otherwise} \end{cases}$$

where $h_i(p_j)$ is the value computed from the i-th hash function over the j-component prefix p_j, and $\oplus$ is the XOR operator. The use of XOR operations significantly speeds up the computation time without impacting hashing properties [SHKL09].

3.3.4 Hash table design

The PBF performs a membership query on content prefixes to find the longest prefix length stored in the FIB. Once the longest prefix length is chosen, the second stage of our name-based LPM algorithm consists of a hash table lookup to either fetch the next hop information or to manage the possible false positives.

Our hash table design is shown in Figure 3.3. The hash table used for our forwarding module consists of several buckets where the prefixes in the FIB are hashed to. In a similar way of the PBF's functions, we keep in mind, for the hash table design, the goal to minimize memory access latency, in order to speed-up the overall processing rate. For this purpose, each bucket is restricted to the fixed size of one cache line such that, for well-dimensioned tables, only a single memory access is required to find and fetch an entry. In case of collisions, entries are stored next to each other in a contiguous fashion up to the limit imposed by the cache line size. We can manage the bucket overflows by chaining with linked lists, but this is expected to be rare if the number of buckets is large enough.

To furtherly improve the lookup performance, our second design goal is to reduce the string matching overhead required to find an entry. Therefore, each entry stores the hash value h of its content prefix. String matching on the content prefix only occurs if there is a match first on this 32-bit hash value. Due to the large output range of the hash function, an error is expected only with small rate: we estimated in Section 2.4.3 the probability of collision for our dataset to be equal to 0.002.

Finally, our last goal is to maximize the capacity of each bucket. For this purpose, the content prefix is not stored at each entry due to its large and variable size. Instead, only a 64-bit pointer p to the prefix is stored. To save space, next-hop MAC addresses are also kept in a separate table and a 16-bit index a is stored in each entry. A 16-bit index i is also required per entry to specify the output line card of a given content prefix. Each entry in the hash table then consists of a 16-byte tuple $\langle h, i, a, p \rangle$, where h is the hash of the content prefix, i is the output line card index, a is index of the next-hop MAC address, and p is the pointer to the content prefix. With this configuration every bucket consists of 8 slots at most. The *bucket size* is the maximum number of slots it can contain, and is denoted by s. If needed, the last slot contains the pointer to the linked list used in case of bucket overflow.

3.3.5 Caesar extensions

In this section, we introduce two Caesar extensions in order to meet increasing space/performance requirements. We can support large FIBs (tens of GigaBytes) by having each line card store only part of the entire FIB and collaborate with each other in a distributed fashion. Caesar's performance can be furtherly improved by offloading large packet batches to a graphics processing unit (GPU). These solutions may introduce additional latency during packet processing and thus are presented here as extensions that can be activated at the operator's discretion.

Large FIB In its original design, Caesar stores a full copy of the FIB at each line card, as commonly done by commercial routers. Although this allows each line card to independently process packets at the nominal rate, it also results in FIB replication and waste of storage resources. For IP prefixes, this is usually not a concern, as a typical FIB's size is around five hundreds thousands entries in the core network. In NDN, however, the FIB can easily grow past hundreds of millions of content prefixes [SNO13] and memory space may become a major issue.

We propose a Caesar extension, analyzed in details in Section 3.3.6.3, that allows multiple line cards to share their FIB entries. We propose to store in each line card only a subset of the original FIB entries such that, overall, Caesar is able to store $\mathcal{N}$ times more entries.

High-speed forwarding The classic solution to increase forwarding speeds in network devices is a hardware update. However, this approach is not scalable due to the high costs both in terms of hardware purchasing and reconfiguration time. While this may be an option for the deployment of edge/core routers with a large set of networking features, such cost is prohibitive for an enterprise router.

In Section 3.3.6.4, we propose an alternative strategy that does not incur such a high cost. Wang *et al.* [WZZ+13] have recently shown that high-speed LPM on content names is possible by exploiting the parallelism of popular off-the-shelf GPUs. As a second Caesar extension, we propose to use GPUs to accelerate packet processing. Currently, commodity GPUs are available in the market at a lower price compared to the upgrade cost. The challenge is then how to efficiently leverage a GPU to guarantee fast name-based LPM. For this extension, we assume that a GPU is connected to each line card using a regular PCIe 16x bus and that it stores the same FIB entries as the line card.

3.3.6 Implementation

We implement our design using the classical differentiation between data and control plane. Our data plane is described in Section 3.3.6.1. We describe the most important features of our control plane in Section 3.3.6.2.

3.3.6.1 Data Plane

The forwarding module is designed as a modular component inside a content router, whose architecture is shown in Section 2.3. We now analyze the data plane of the forwarding module. Caesar's forwarding module is responsible for forwarding packets received by the line cards. Similarly to a common IPv4 data plane, we may identify the most important processes.

Packet input: As a packet is received from the SPF+ 10GbE external interface, it is stored in the off-chip DRAM of the line card via DMA. A hardware scheduler then assigns the packet to one of the available cores for processing.

Header parsing: In Section 2.4.2 we show our definition of Interest and Data packets' headers. One of the operation that the forwarding module has to perform is to detect if the incoming packet is a regular IP packet or an NDN packet (by checking the protocol field in the IPv4 header, or by checking the specific type field in the NDN header). Once an NDN Interest packet is detected, the content name is extracted and it is used to perform the LPM on the FIB table: the name-based header contains pointers to each component, which are then stored in

the L1 cache in order to improve memory access speed. Otherwise, regular packet processing is performed, i.e. LPM on the destination IP address.

Name-based LPM: If such a header is found, our name-based LPM algorithm is used. The size of each PBF block is set to one cache line, or 128 bytes in our architecture (cf. Section 3.4).

Since Caesar takes advantage of hardware co-processors in the NPU, the forwarding module may exploit such those functionalities to ensure fast hashing calculations. In particular, the $k+d-1$ seed hash values are computed using CRC32 hardware functions, whereas the remaining $(k-1)(d-1)$ hash values are computed from XOR operations. In case of a match in the PBF, the content prefix is looked up in the hash table stored in the off-chip DRAM to determine its next hop information, or to rule out false positives. Each table entry has a fixed size of 16 bytes, which, for a bucket of 128 bytes, results in a maximum of 7 entries per bucket in addition to the 64-bit pointer required by the linked list. We dimension the hash table so to contain 10M buckets; it follows that the hash table requires 1.28 GB to store the buckets and 640 MB to store the content prefixes, for a total of 1.92 GB.

Switching: The LPM algorithm returns the index of the output line card and the MAC address of the next hop for a packet. The source MAC address of the packet is then set to the address of the backplane interface, and its destination MAC address is set to the address of next hop. Finally, the packet is placed into a per-core output queue in the backplane interface and waits for transmission. Each NPU core has its own queue in the backplane interface to enable lockless queue insertions and avoid contention bottlenecks. Once transmitted over the backplane, the packet is received by the switching fabric, and regular L2 switching is performed. The packet is then directed to the output line card over the backplane once again.

Packet output: Once received by the backplane interface of the output line card, the packet is assigned to a NPU core and the source MAC address is overwritten with the address of the SPF+ 10GbE interface. The packet is then sent via DMA to this interface for external transmission.

3.3.6.2 Control Plane

The Control Plane is out of the scope of this thesis, but for the sake of clarity we may still analyze the most important features. The forwarding module work in conjunction with a Control plane, that is responsible for periodically computing and distributing the fowarding information base (FIB) to line cards. These operations are performed by the route controller, a central authority that is assumed to participate in a name-based routing protocol [HAA+13]

to construct its routing information base (RIB). The RIB is structured as a hash table that contains the next hop information for each reachable content prefix.

The FIB is derived from the RIB and is composed of the PBF and the prefix hash table. To allow prefix insertion and removal, the route controller maintains a mirror counting PBF (C-PBF). For each bit in the PBF, the C-PBF keeps a counter that is incremented at insertions and decremented at removals. Only when a counter reaches zero the corresponding bit in the PBF is set to 0. The C-PBF enables prefix removal while avoiding to keep counters in the original PBF, which saves precious L2 cache space.

The C-PBF is updated on two different timescales. On a long timescale (i.e. minutes), the C-PBF is recomputed from the RIB with the goal to improve prefix distribution across blocks. On a short timescale (i.e. every insertion/removal) the C-PBF is greedily updated. When inserting a new prefix, additional expansions are performed on blocks that exceed the false-positive threshold. When removing a prefix, block merges are postponed until the next long-timescale update.

The content prefixes stored in the i-th block of the PBF are hierarchically organized into a prefix tree to (1) easily identify the length at which the threshold TV_i is exceeded, and (2) efficiently move prefixes during block expansions with a single pointer update operation. The prefix tree of each block is implemented as a left-child right-sibling binary tree for space efficiency.

3.3.6.3 Distributed Forwarding

To share a large FIB among line cards, we implement a forwarding scheme where LPM is performed in a distributed fashion. The idea is for each packet to be processed at the line card where its longest prefix match resides, i.e. not necessarily the line card that received the packet. A fast mechanism must then be in place for each received packet to be directed to the correct line card for LPM. For this extension, the following modifications to Caesar's control and data planes are required.

Control plane: The route controller now has to compute a different FIB per line card. Each content prefix p in the RIB is assigned to a line card LC_i, such that $i = g(p_1) \bmod \mathcal{N}$, where $g(p_1)$ is the hash of the subprefix p_1 defined by the first component of p. The rationale here is the same used in the PBF for block selection (cf. Section 3.3.1); by distributing prefixes to line cards based on their first component, it is possible for an incoming packet to be quickly forwarded to the line card where its longest prefix match resides.

In addition to distributing the FIB, the route controller also maintains a *Line card Table* (LT) containing the MAC address of the backplane interface of each line card. The LT is distributed to each line card along with their FIB, and serves two key purposes.

First, the LT is used by each line card to delegate LPM to another card (see below). Second, the LT allows the router controller to quickly recover from failures. With distributed forwarding, the failure of a line card may jeopardize the reachability to the prefixes it manages. We solve this issue by allowing redirection of traffic from a failing line card to a backup line card. Once Caesar detects a failure at a line card LC_i, the route controller sends the FIB of LC_i to one of the additional line cards pre-installed and updates the LT to reflect the change. The updated table is then distributed to all line cards to complete the failure recovery.

Data plane: Upon receiving a packet with content name x, an available NPU core computes the target line card LC_i to process the packet, with $i = g(x_1) \bmod \mathcal{N}$. If LC_i corresponds to the local line card, then the regular flow of operations occurs, i.e. header extraction, name-based LPM, switching, and forwarding (cf. Section 3.3.6). Otherwise, the destination MAC address of the packet is overwritten with the address of the backplane interface of LC_i fetched from the LT, and the packet is transmitted over the backplane. LPM then occurs at LC_i and the packet is sent once again over the backplane to the output line card for external transmission.

Distributed forwarding imposes two constraints as tradeoffs for supporting a larger FIB. First, it introduces a short delay caused by packets crossing the backplane twice. Second, extra switching capacity is required. In the worst case, i.e. when a packet is never processed by the receiving line card, the switch must operate twice as fast at a rate $2\mathcal{N}R$, where R is the rate of a line card, instead of $\mathcal{N}R$. Nonetheless, as showed in [IM03], it is possible to combine multiple low-capacity switch fabrics to provide a high-capacity fabric with no performance loss at the cost of small coordination buffers. This is a common approach in commercial routers, e.g. the Alcatel 7950 XRS leverages 16 switching elements to sustain an overall throughput of 32 Tbps.

3.3.6.4 GPU Offloading

Our Caesar extension to accelerate packet forwarding makes use of a GPU. First, a brief background on the architecture and operation of the NVIDIA GTX 580 [nG] used in our implementation is provided. Then, a discussion on our name-based LPM solution using this GPU is presented.

The NVIDIA GTX 580 GPU is composed of 16 streaming multiprocessors (SMs), each with 32 stream processors (SPs) running at 1,544 MHz. This GPU has two memory types: a large, but

slow, *device memory* and a small, but fast, *shared memory*. The device memory is an off-chip 1.5 GB GDDR5 DRAM, which is accelerated by a L2 cache used by all SMs. The shared memory is an individual on-chip 48 KB SRAM per SM. Each SM also has several registers and a L1 cache to accelerate device memory accesses.

All threads in the GPU execute the same function, called *kernel*. The level of parallelism of a kernel is specified by two parameters, namely, the number of *blocks* and the number of *threads per block*. A block is a set of concurrently executing threads that collaborate using shared memory and barrier synchronization primitives. At run-time, each block is assigned to a SM and divided into *warps*, or sets of 32 threads, which are independently scheduled for execution. Each thread in a warp executes the same instruction in lockstep.

An application that aims to offload processing to a GPU works as follows. First, the application copies the data to be processed from CPU to device memory. Then, the application launches a kernel; the kernel reads the input data from device memory, performs a desired operation and then write results back to device memory. Finally, the application copies results back from device memory to CPU's memory. GPUs suite computation-intensive applications because of their extreme thread-level parallelism as well as latency hiding capabilities.

Name-based LPM: We introduce few modifications made to the LPM algorithm to achieve efficient GPU implementation. Due to the serial nature of Caesar's processing units, the original algorithm uses a PBF to test for several prefix lengths in the same filter. However, to take advantage of the high level of parallelism in GPUs, a LPM approach that uses a Bloom filter and hash table per prefix length is more efficient. Since large FIBs are expected, both the Bloom filters and hash tables are stored in device memory.

For high GPU utilization, multiple warps must be assigned to each SM such that, when a warp stalls on a memory read, other warps are available waiting to be scheduled. The GTX 580 can have up to 8 blocks concurrently allocated and executing per SM, for a total of 128 blocks. Content prefixes are assumed to have 128 components or less, and thus we have one block per prefix length in the worst case. Since such a large number of components is rare, we allow a higher degree of parallelism with multiple blocks working on the same prefix length. In this case, each block operates on a different subset of content names received from a line card.

To keep track of the prefix lengths available in the FIB, we use a *prefix length mask* (PLM), a bit array where the i-th bit indicates the presence of at least one prefix with i components in the FIB. We have $size(PLM) \leq 128$, matching the highest number of components that we can handle at the maximum level of parallelism. This size is chosen after finding the maximum prefix length stored in the FIB, at compile-time. When matching a content name with d components, the PLM is first checked from position d to 1. We preload the PLM to shared

memory to speedup the masking of a content name. Our algorithm receives as input arrays $\overline{B}$, $\overline{H}$, and $\overline{C}$ that contain the Bloom filters, hash tables, and content names to be looked up, respectively. The kernel identifies the length of longest prefix in the FIB that match each content name $c \in \overline{C}$ and stores it in the array L, which is then returned to the line card. All these arrays are located in the device memory.

The algorithm works as follows: each block computes the index of the prefix length it is responsible to perform lookups for. Of course, many blocks may be used per prefix length, and only in the worst case (i.e. at least one prefix with 128 components) a block is associated to a single prefix length. All blocks apply the mask to each input content name based on the PLM, and prepare the execution of the LPM algorithm, which is performed in parallel by every thread in the warp. Each iteration of the thread consists in loading a different content name and performing a Bloom filter lookup. If a match is found, a further hash table lookup is performed to check the possible false positive. If an entry is found in $\overline{H}$, the prefix length is written to the array L. The final operation is an atomic check across all SMs, to confirm that the LPM for that specific content name is realized and only the longest prefix indexed by the array L is returned.

This iteration continues up to the end of the batch of content names $\overline{C}$. The input names of our GPU algorithm are batched, that is a line card that wants to offload some processing has to buffer a subset of the incoming content names, and send the whole batch to the GPU all at once; the output of the GPU processing is transfered with the same approach as soon as the results of all lookups are computed. An extended version of this algorithm, which can perform not only LPM searches but also more generic tuple lookups, is published in [VLZL14].

3.4 Evaluation

This section experimentally evaluates Caesar's forwarding module. First, we describe our experimental setting and analyze the Prefix Bloom Filter. Then, we evaluate both basic design and its enhancements, namely distributed forwarding and GPU offload.

3.4.1 Experimental setting

We assume that Caesar's line cards work in half-duplex mode, two as input line-cards and two as output line cards. We equip Caesar with our forwarding module and connect its line cards

to a commercial traffic generator[4] equipped with 10 Gbps optical interfaces via optical fibers. Then we measure its forwarding throughput and latency. The throughput is the fastest rate of packets that are forwarded without packet losses averaged over a 60 seconds time-frame. It is measured by generating traffic at 10 Gbps to every input line-card and by increasing the packet size until there are no losses for 60 seconds. The latency is described by the minimum, maximum and average packet latency over a 60 seconds time-frame. We extend the traffic generator to support the NDN-like packet format described in Section 1.4.1, i.e. regular IP packets with an additional name-based header.

The elements stored in the FIB and requested content names derive from the reference workload shown in Section 2.4.3 (more precisely, Figure 2.3). We assume incoming content names have length $\nu = 42$ bytes. It is useful to define a "distance" t between content prefixes as the difference, measured in number of components, between the incoming content name and the longest prefix stored in the FIB. For instance, if `a/b/c/d/e` is the incoming content name, and `a/b` is its longest prefix, then t is equal to 3. In our workload the average distance t between content prefixes and input content names is equal to 2. This value represent a metric to detect how many trials should be performed on average before finding a FIB match.

The synthetic workloads are generated from the reference workload by varying the following parameters: i) *distance t*, which impacts the number of hash values to be computed as well as the number of potential PBF/hash table lookups; ii) *number of prefixes*, which defines the size of the FIB and thus its memory footprint and access speed ; iii) the number of prefix sharing the *first component*, which impacts the amount of prefixes stored per block and thus the PBF false positive.

3.4.1.1 Hash table dimensioning and analysis

We dimension our hash table to contain all the prefixes of the reference workload described in Section 2.4.3. The choice of the hash table size takes into account a parameter α that is the *load factor*, defined as the ratio between the number of elements stored in the hash table and the number of buckets it contains. Given that β denotes the number of buckets in our table, and n is the number of stored elements, we have $\alpha = \frac{n}{\beta}$. We remind that in our design, as described in Section 3.3.4, every bucket contains $s = 8$ slots.

The load factor is related to the access speed and memory usage of the hash table: for $\alpha \ll 1$, the hash table is almost empty, the access speed may increase at the cost of a memory waste. For $\alpha \gg 1$, the table is overloaded, and many slots per bucket are used: this increase the

[4]Spirent SmartBit 6000B
`http://www.smartechconsulting.com/SMB-6000B-Spirent-Smartbits-SMB6000B-12-Slot-Desktop-Chassis`

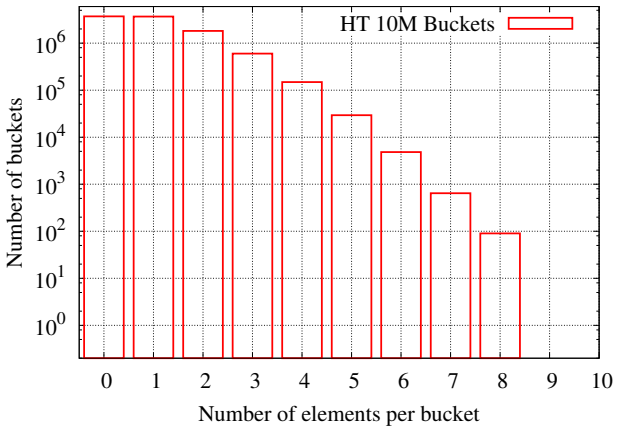


Figure 3.4: *Number of prefixes per bucket in the FIB. Since at most 8 slots are occupied, there is no bucket overflow.*

average lookup time because many slots might have to be checked in order to find a match. Our design choice is to set $\beta = n$, that is, load factor $\alpha = 1$. Since the size of a bucket is one cache line, it follows that the hash table requires 1.28 GB to store the buckets. Next-hops storage, as well as the content prefixes and the MAC address information, require 640 MB of memory, for a total amount of 1.92 GB.

When the FIB is populated, every prefix is stored in a slot of a bucket, selected as described in Section 3.3.4. For each prefix p, we calculate the value $h(p)$ using the CRC32 function, and select the bucket at position $h(p)$ *mod* β. Then, we choose the first available slot where the prefix is eventually stored. If the bucket overflows, the last slot is used as a pointer to an appended linked list of entries. Figure 3.4 shows the distribution of the number of prefixes per bucket, calculated over the reference workload with $\beta = 10$M and $s = 8$. We note that no bucket has more than 8 elements stored: this is important, since it shows that we do not need to manage the bucket overflows by means of the extra linked list. Finally, 99% of the buckets requires less than 4 slots occupied, and about 37% of the buckets are empty.

3.4.1.2 PBF dimensioning and analysis

The first step for a correct choice of the PBF's parameters (number of blocks, size of the blocks, number of cache lines) can be made after performing a dimensioning. This section analyzes the prefix Bloom filter and derives a set of parameters used for the evaluation of the forwarding module.

PBF block size We set the number of hash functions $k = 2$, in order to minimize the number of seed hash values to be computed. As shown in Section 3.4.2 at page 72, the calculation of

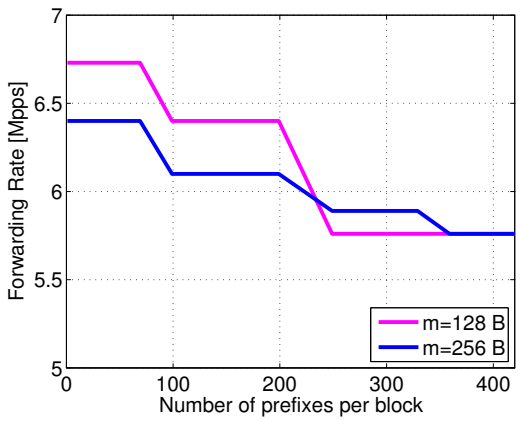


Figure 3.5: *Throughput as a function of the number of prefix per block.*

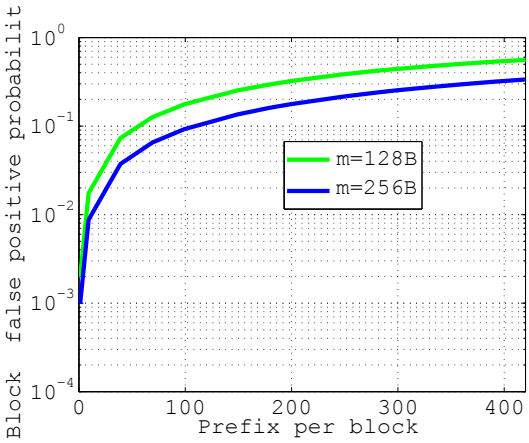


Figure 3.6: *False positive as a function of the number of prefix per block.*

seed hash values is an expensive operation in terms of computation time. It follows that the generation of additional seed hash values (i.e., $k > 2$) while reducing the false positive probability, it significantly degrades the throughput of the system.

Figure 3.5 analyzes the throughput of our forwarding module as a function of the number of prefixes stored in every PBF block assuming block size of one/two cache lines (i.e. 128/256 B). For this analysis we consider a synthetic workload derived from the reference one where we vary the number of prefixes sharing a common first component. We observe that when the number of prefixes per block is low, i.e.less than 200 prefixes per block, increasing the block size reduces the overall throughput (measured in Mpps - Million packets per second). This happens because a block size larger than a cache line requires additional memory accesses to DRAM at each LPM operation.

As shown in Figure 3.6 the advantage of a larger block size is that it provides a lower false positive probability for the same amount of prefixes per block. It follows that the number of prefixes per block increases, the lower false positive ratio for bigger blocks translates into actual throughput improvements, i.e. when the are more than 200 prefixes. Nevertheless this set of results shows that a block of a single cache line with a threshold for expansion of less that 100 prefixes per block guarantee the highest throughput. Therefore, for the rest of the evaluation we set the the block size b=128B and the expansion threshold value TV=75 prefixes.

PBF size Figure 3.7 reports the throughput as a function of the PBF size M for the reference workload. We observe that the throughput sharply increases with the PBF size for small value of M. Then, after a certain size the throughput remains constant. As shown in figure 3.8 for small value of M a larger PBF size significantly reduces the false positive probability. On the contrary, for large value of M to increase the PBF size only slightly reduces the false

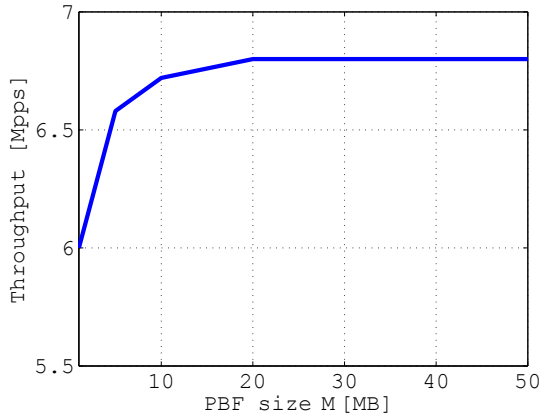


Figure 3.7: *Throughput as a function of the PBF size.*

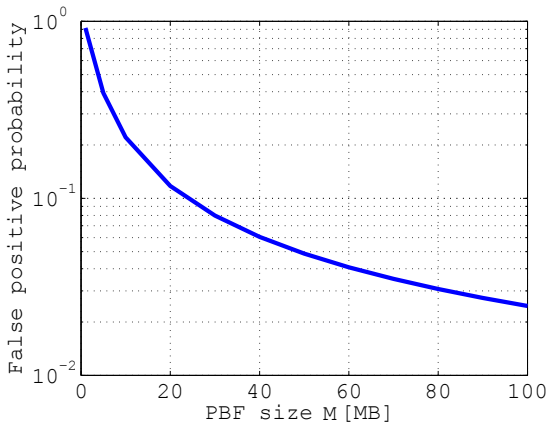
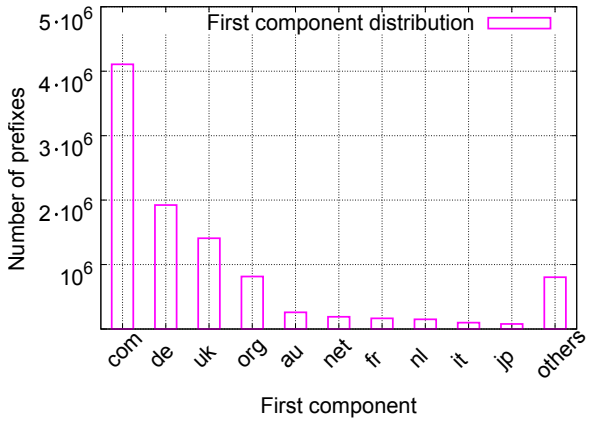
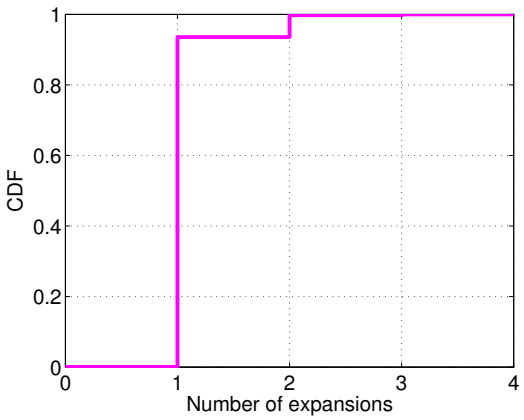


Figure 3.8: *False positive as a function of the PBF size.*



(a) *First components in the reference workload*

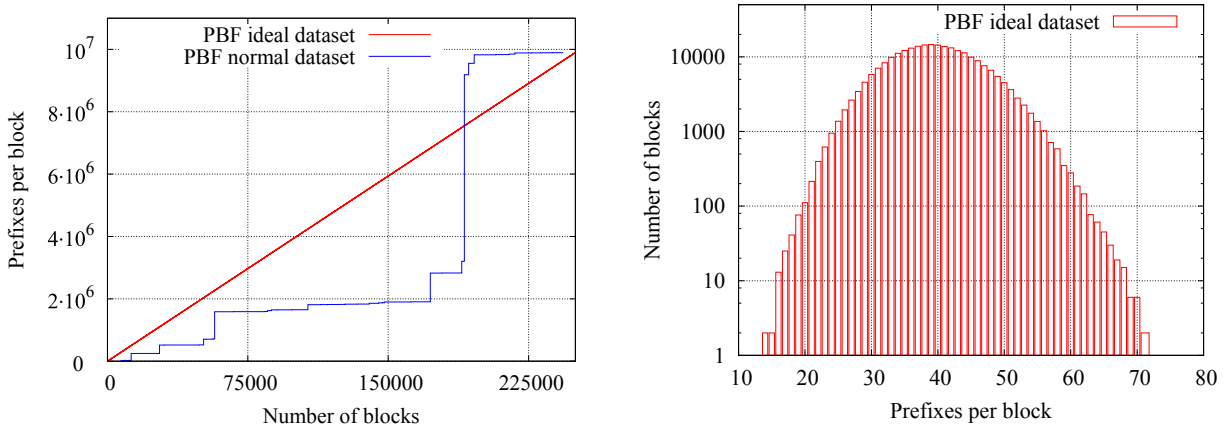


(b) *Number of expansion per prefix. Reference workload, threshold $TV = 75$.*

Figure 3.9: *Analysis of the reference workload*

positive probability and does not provide any actual throughput increase. We therefore set the PBF size to the minimum value required to guarantee the maximum forwarding throughput, i.e. $M = 32$ MB.

Load balancing In Section 3.3.2 we discussed about the expansion mechanism needed when the first component distribution is very skewed. Figure 3.9a shows indeed that in the reference workload more than 90% of the dataset shares only 10 different first components. The expansion is therefore mandatory, since few blocks will be overloaded (e.g. the block of the `"com/"` component will have to deal with more than 6M prefixes). Before to present experimental results we report in Figure 3.9b the CDF of the number of expansions required per prefix in the reference workload assuming the parameters derived in the previous section, i.e. k=2, m=128B,



(a) *Cumulative prefixes distribution of the reference and the ideal workload*

(b) *Density distribution of prefixes per block in the ideal workload*

Figure 3.10: *Analysis of the distribution of the first component in the ideal and the reference workload.*

and TV=75. We observe that 95% of prefixes only requires one expansion, while only 1% requires 4 expansions, the maximum number of expansions found in the reference workload.

However, throughout our evaluation we would like to test as well an ideal case, where expansion is never required. We thus produce a synthetic "ideal" workload by adding a random number to the first component, so that the first components spread across the PBF blocks. We show this in Figure 3.10a and 3.10b. The former is a comparison between the reference workload and our synthetic ideal workload; it shows the cumulative distribution of prefixes w.r.t. the PBF blocks. We observe that the reference workload is represented by a step function, where every step consists of the increment due to the few first components. On the contrary, the ideal workload is very uniform, producing a straight line of the cumulative distribution of prefixes. Finally, Figure 3.10b shows the distribution of prefixes per block in the PBF with the ideal workload: we observe that all the blocks in the ideal workload contain less than 75 prefixes, which is our threshold value; this grants that no expansion is ever needed. On average, every block is charged with 40 prefixes.

3.4.2 Performance evaluation

This section provides the results of the performance evaluation of our forwarding module. We consider three configurations of our LPM algorithm: i) *PBF*, where the PBF is used without the expansion; ii) *PBF exp*, where the PBF is expanded; iii) *NoBF*, where the PBF is not used and all matching hash tables are looked up. We also report results for an ideal case, *PBF ideal*, where the PBF is used without expansion and all prefixes have a different first component, i.e. content prefixes are uniformly distributed over blocks.

	PBF ideal	PBF exp	PBF	NoBF
Throughput Match [Mpps]	6.73	6.63	6.34	6.31
Throughput No Match [Mpps]	7.53	7.5	6.31	5.27
Min latency [μs]	6.6	6.6	6.6	6.6
Avg latency [μs]	7.3	7.5	7.7	7.7
Max latency [μs]	9.9	10.6	12.2	12.3

Table 3.1: *Throughput (Mpps - Million packets per second) and latency (μs) under the reference workload $n = 10M$.*

In the remainder of the section, we first evaluate the forwarding module's performance and then we present a sensitivity analysis where we vary parameters from such workload.

Reference workload

We start by measuring Caesar's throughput when using our forwarding module activating a single 10 Gbps input line-card. For the LPM algorithm, we differentiate between PBF, PBFexp and NoBF, as described above. Finally, we assume the reference workload.

Table 3.1 summarizes the results derived from this set of experiments. We first focus on the throughput Caesar achieves assuming the PBF exp. The table shows that Caesar supports up to 6.63 Mpps (Million packets per second) when all incoming content names match at least a FIB entry ("Match"), and up to 7.5 Mpps when none of the incoming packets match a FIB entry ("No Match"), in which case the packet is forwarded to a default route. 6.63 Mpps translates to a minimum packet size of 188 Bytes for a 10 Gbps line card at full rate. We verified that this result extends to the full router, i.e. the forwarding module handles about 20 Gbps in input and 20 Gbps in output with 188 Bytes.

Table 3.1 also shows that the Bloom filter expansion mechanism, only causes a 2% throughput drop with respect to the ideal case (PBF ideal). This drop is caused by additional memory accesses and complexity. The PBF without expansion has a performance gap of 4% with respect to PBF exp because of the higher false positive rate. Finally, the table shows limited throughput benefits in the usage of a PBF rather than multiple hash table accesses when all incoming content requests match at least a FIB entry (about 5% improvement). This happens because of the simplicity of the reference workload where the distance between content requests and content prefixes is low, on average only 2 components. It follows that in the NoBF case, only two additional hash table accesses are required. We will investigate more adversarial workloads in the upcoming subsection. Nevertheless, the table shows clear benefits of the PBF when none of the incoming packets match a FIB entry, e.g. about 30% performance increase. This result suggests that our LPM algorithm is robust to DoS attacks, where an attacker generates non-existing content names to slow down a content router.

We now focus on packet latency. Table 3.1 indicates that PBF exp provides a slightly lower latency than both PBF and NoBF, on average; again, this is due to the simplicity of the reference workload, and the packet latency is dominated by the latency due to the switching operation. However, even under the reference workload, PBF exp reduces the maximum latency by 15%. The maximum latency is observed for packets whose content names have many components (e.g. 12); in this case the LPM latency becomes more significant and PBF with the expansion mechanism provides benefits. We also notice that the expansion mechanism only causes a 2-6% latency increase with respect to the ideal case in the average/maximum case respectively.

We now dissect the performance bottlenecks that occur in the forwarding operation. We observe the network processor cores and the software operations are the main bottlenecks of the system [DEA+09]. First, if we run the system without performing the LPM operation, every input line-card can process 15.6 Mpps. Second, the amount of cycles consumed by every core per packet does not depend on the packet throughput. We thus investigate how CPU cycles are used in Table 3.2. Specifically, we report: i) the number of cycles required for every major software operation performed by every core of an input-line card; ii) the total number of cycles per operation under the reference workload, differentiating between PBF ideal, PBFexp, PBF, and NoBF. Overall, the hash-table lookup is the most expensive operation. Content name hashing also has a significant impact when many hash values are computed: this happens when the distance t between content requests and content prefixes is large (cf. Figure 3.11). Also a PBF lookup is computationally expensive when more than one blocks is loaded: this is highlighted by the PBF exp as a consequence of the expansion operations. Finally, the I/O operation have a non negligible impact on the performance but,as expected, they are not affected by the considered LPM algorithm.

Sensitivity analysis

We now vary several parameters of the reference workload in order to evaluate their impact on Caesar's forwarding module. We start by analyzing the impact of the "distance" t between content prefixes and input content names, i.e. the difference of their lengths as number of components; this is the most important parameter as it reflects the complexity of the LPM. To this goal, we generate several synthetic workloads where the average number of components

	Total	I/O	Hashing	HT lookup	PBF lookup
Atomic operation	-	371	107	351	129
PBF ideal	1499	371	363	484	294
PBF exp	1783	371	440	485	357
PBF	1849	371	440	756	357
NoBF	1948	371	462	1048	-

Table 3.2: *Cycles required for every operation.*

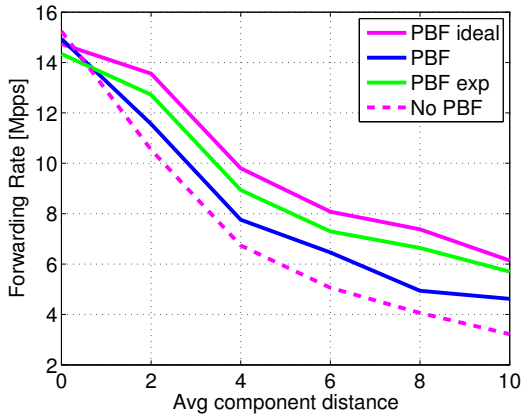


Figure 3.11: *Throughput as a function of average component distance t.*

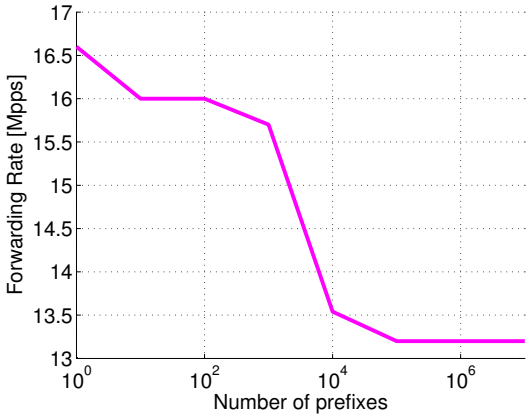


Figure 3.12: *Throughput as a function of the number of prefixes.*

per content name d varies between 2 and 10 components, i.e. from $t = 0$ (equivalent of exact match) to $t = 10$ (highly adversarial workload) assuming unchanged average prefix length of 2 components as in the reference workload.

Figure 3.11 shows the forwarding throughput as a function of t distinguishing between PBF ideal, PBF, PBFexp and NoBF. Overall, the throughput decreases as t increases; this happens because the number of hash values to be computed increases linearly with t. Compared to NoBF, the throughput improvements provided by the usage of PBF increase with t; when $t = 10$, PBF ideal and PBFexp almost double the throughput achieved by NoBF. Overall, PBFexp introduces a penalty when t is small, which is then absorbed as t increases, while also the PBF without expansion provides significant benefits.

We now investigate the impact of the FIB size, n content prefixes, on the forwarding module in Figure 3.12. Overall, the figure shows that the throughput follows a step function: when $n = 1$ the throughput is about 8.3 Mpps; when $1 < n < 1,000$, the throughput is 8 Mpps, while for $n > 1000$ it suddenly drops to 6.6 Mpps. When $n = 1$ the prefix is permanently stored in the L1 cache of every core. Then, when the number of prefix is small, the network processor efficiently stores the prefixes in the L2 cache; after 1,000 prefixes, the L2 caches is also exhausted and prefixes are placed in the off-chip DRAM, which causes such throughput drop. After the 1,000 prefixes threshold, the throughput is almost constant: this indicates that the amount of prefixes that the FIB may support is only limited by the amount of off-chip DRAM. To store more prefixes, it would be sufficient to add more memory while the speed would not be impacted. This is typically the case on edge/core routers where the amount of DRAM available is in the order of tens of GBytes. Implementing this forwarding module on such platforms would allow to store one-two order of magnitude more prefixes while performing line-rate forwarding.

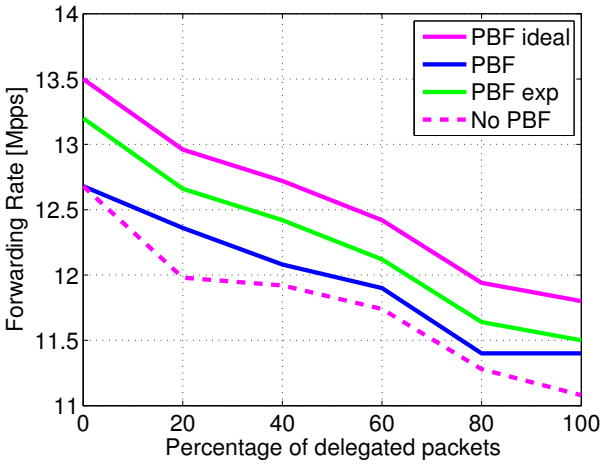


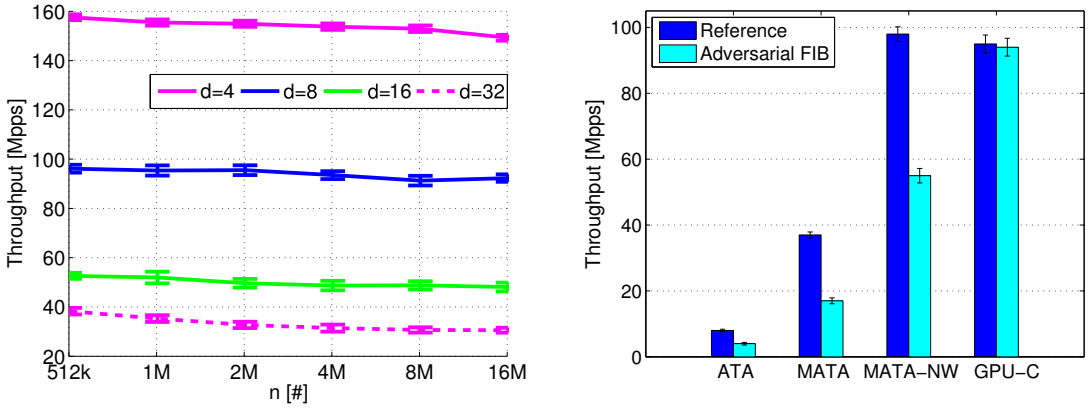
Figure 3.13: *Throughput as a function of ρ.*

3.4.3 Distributed Processing

This section evaluates distributed forwarding, a mechanism to increase the number of prefixes stored in the FIB linearly with its number of input line cards. We populate each input line-card with a disjoint set of 10 M prefixes, and we assume the reference workload. Since Caesar mounts a 10 Gbps switch, we limit the traffic injected to each input line card of its forwarding module to 5 Gbps; this is the theoretical maximum that such switch can sustain when distributed forwarding is used for 100% of the packets, i.e. two switching operations per packet.

Figure 3.13 shows the forwarding throughput of each line card as a function of the fraction of packets distributed to the other line card for LPM, ρ in the following. Overall, the throughput decreases as ρ increases. While a line card still processes the same amount of packets (i.e. the ones received and processed locally, and the ones received from the other line card), packets that are not processed locally require additional operations, namely packet dispatching, and MAC address rewriting. In the worst case, when ρ=100%, these operations account for a throughput drop of 15%.

We also estimate the impact of distributed forwarding on packet latency, assuming ρ=100%. We find that the average and minimum latency increase by 50% compared to basic Caesar: as previously discussed, minimum and average latency mostly derives from switching latency which is hereby doubled. The maximum latency grows instead by about 30% under distributed forwarding: this happens when LPM latency overcomes the switching latency, i.e. in presence of long content names. In any case, the additional latency remains in the order of μs and it is thus tolerable even for delay sensitive applications.



(a) *n=0.5M:16M ; Equally distributed.* **(b)** *Comparison with [WZZ+13]*

Figure 3.14: *Evaluation of forwarding module enhancements with the GPU offloading.*

3.4.4 GPU Off-load

This section investigates the throughput speedup that GPU offloading provides to the forwarding process. We assume a line card offloads a batch of 8K content names to our GPU, NVIDIA GTX 580 [nG]. Note that such batch is large enough to ensure high GPU occupancy.

We first measure how many packets per second the GPU can match as a function of the number of rules n in the FIB (cf. Figure 3.14a). Such throughput refers only to kernel execution time; we omit the transferring time between line card and GPU, and vice-versa, in order to be comparable with [WZZ+13] and not limited by the PCIe bandwidth problem therein discussed. We generate several synthetic FIBs where the number of content prefixes grow exponentially from half a million to 16 million prefixes, the maximum number of prefixes that fit in the GTX 580's device memory; we also vary the number of unique content prefix component lengths d between 4 and 32 components. Finally, we assume that content names have 32 components and that content prefix lengths are equally distributed, e.g. when $d = 4$ a quarter of the prefixes have a single component.

Figure 3.14a shows two main results. First, the throughput is mostly independent from the number of prefixes n: overall growing the FIB size from half a million to 10 million prefixes causes less than a 10% throughput decrease. Second, the throughput largely depends on the number of unique component lengths (d) in the FIB. For example, when $n = 16$ M increasing d from 4 to 32 components reduces the throughput by 5x, from 150 to 30 Mpps.

We now compare our implementation with the work in [WZZ+13], which also explores the usage of GPU for name-based forwarding. Fortunately, they open-sourced their GPU code which allows us to perform a fair comparison with our implementation. The key idea of the

work in [WZZ+13] is to organize the FIB as a trie as done today for IP. They thus introduce a character trie which allows for LPM on content names. Then, they introduce three optimizations, namely the Aligned transition array (ATA), the Multi-ATA (MATA) and the MATA with interweaved name storage (MATA-NW), which leverage a combination of hashing and the hierarchical nature of the content names to realize efficient compression and lookup.

Figure 3.14b compares the performance of our kernel with the kernels proposed in [WZZ+13], namely ATA, MATA and MATA-NW by running their code on our GPU. For such comparison, we use the reference workload, where $d \approx 3$, as well as a more adversarial workload where $d = 8$. We refer to this adversarial workload as "adversarial FIB".

Compared to the results presented in [WZZ+13], we measure less than half the throughput for ATA, MATA and MATA-NW; this is no surprise since our GPU has half the cores than the GTX 590, the GPU used in [WZZ+13]. The figure also shows that the throughput measured for Caesar's Forwarding Module, about 95 Mpps, matches the results from the synthetic traces when $d = 8$ and $n = 10$ M (Figure 3.14b). MATA-NW is slightly faster than our module, 100 versus 95 Mpps, assuming the reference workload. This happens because MATA-NW exploits the fact that content prefixes in the FIB are very short, e.g. 2 or 3 components, to reduce LPM to mostly an exact matching operation. Instead, our algorithm for LPM does not make any such assumption; this design choice makes it resilient to more diverse FIBs at the expense of a performance loss with more simplistic FIBs. Such feature is clearly visible under the adversarial FIB workload: in this case, the forwarding module of Caesaris twice as fast as MATA-NW.

3.5 Conclusion

The first step to match the increasing request for content distribution functionalities, in the place of the original host-to-host communication primitives, consists in building a content router capable of performing name-based forwarding.

Since Future Internet architectures are expected to depart from a host-centric design to a content-centric one, routers must operate on content names instead of IP addresses, at a processing speed that should be comparable with current routers' performance.

The address space explosion, both in number of content prefixes and their length, expected to be on the order of tens of bytes as opposed to 32 or 128 bits for IPv4 and IPv6 respectively, might have been a serious issue for the development of such these functionalities in high-speed equipment.

In this chapter we fill such a gap, by proposing the design and implementation of a forwarding

module as a part of a content router, Caesar, capable of forwarding packets based on names at wire speed. Caesar's forwarding capabilities advance the state of the art in many ways.

First, it introduces the novel prefix Bloom filter (PBF) data structure to allow efficient longest prefix matching operation on content names. Second, it is fully compatible with current protocols and network equipment. Third, it supports packet processing offload to external units, such as graphics processing units (GPUs), and distributed forwarding, a mechanism which allows line cards to share their FIBs with each other. Our experiments show that the forwarding module may sustain many 10 Gbps links, and a FIB with 10 million content prefixes. This is about two orders of magnitude compared to the state-of-the-art larger forwarding tables (BGP) but with a rate that is equivalent to a edge-network high speed router.

We also show that the two proposed extensions allow our module to support both a larger FIB and higher forwarding speed, with a small penalty in packet latency.

Chapter 4

PIT module

This chapter presents the design, implementation and evaluation of the Pending Interest Table (PIT) module of an NDN-based content router. PIT's main role is implementing the symmetric routing feature of NDN, and preventing the creation of loops of Interest packets and duplicates. NDN's communication model (cf. Background Sec. 1.4.1.2) proposes to aggregate Interest packets, or content requests, when they are addressed to the same content name. To achieve these goals, the PIT keeps track of what content is requested and from which linecard's interface: all the unserved requests are stored for a tunable period of time, resulting in a soft-state storage at every node in the network.

The PIT is designed and realized as a module of our content router, Caesar. Content names stored in the PIT module follows the usual scheme proposed by NDN (cf. Section 2.4.2 and Section 3.1). In the PIT content items are not split in their hierarchical components: requests are stored using the full name as identifier, and are then looked-up with an exact-match algorithm. For example, for the packets `/fr/inria/thesis.pdf/packet1` and `/fr/inria/thesis.pdf/packet2`, where the delimiter is "/", we do not exploit the common prefix as done in Section 3.3, and they are considered as different elements.

The Chapter is organized as follows: in Section 4.1 we describe the PIT, the goals and the features of our module. Then we explore the design space in Section 4.2. Our design and the related implementation for a wire-speed Pending Interest Table is shown in Section 4.3. We extensively evaluate our PIT in Section 4.4, using the Caesar prototype coupled with a commercial traffic generator and both synthetic and real traces for content requests. Finally a summary of our main results is shown in Section 4.5.

Among the main findings of this Chapter, we show that a PIT whose size is in the order of 10^6 entries can work at a wire-speed of tens of Gbps. We also highlight and validate by means of

experiments the differences between different PIT placements.

4.1 Description

The Interest propagation, which creates a "breadcrumb routing" mechanism (cf. Background Sec. 1.4.2), is enabled thanks to the state that every NDN-router keeps in their PIT module: equivalently, PIT entries are the breadcrumbs which actually realize symmetric routing. These breadcrumbs are also used to aggregate Interests for the same chunk, naturally realizing multicast at the network layer. The Pending Interest Table is a data structure that is populated when Interest packets reach one line card; this is done in order to provide a soft-state of the requested content that are still to be served. When Data packets follow their way through the content requester, they eventually consume their matching PIT entries. In order to prevent loops and duplicates, Interest packets contain a random nonce value: the PIT is able to detect when the same nonce is received twice, and drops the packet without performing any additional instruction.

PIT operations Three operations can be performed on the PIT: insert, update and delete. When a new Interest is received, PIT should extract the content name from the packet, and check if an entry associated with that name is located in the PIT. It no such entry is present, a PIT entry is created and added to the table. Otherwise, an update operation is performed. PIT should verify at first whether the nonce carried in the packet is present in the entry's list of nonces, in which case a loop is detected and the packet is dropped. Then, it updates the entry's list of interfaces, adding the interface from where the Interest has been received, if not present. The delete operation can occur when a timer expires, or when a Data packet is received. In both cases, PIT performs a lookup to identify the correct entry, and it then removes the item from the table.

Large state The PIT module must support a potentially large state. In fact, the maximum number of elements to be stored in the PIT in the worst case depends on the transmission rate of the router's input interfaces and the entries' deletion time (that may be considered as the time taken by requested data packets to reach the PIT in the backward propagation) with the formula $n_{MAX} = \lambda_{in} \cdot T_{MAX}$ (following from the Little's law [Lit61], where λ_{in} is the input rate, and T_{MAX} is the entries' deletion time). Whether typical latency in CDNs [JCDK01] belongs mostly to the range [100, 250]ms, authors of [YMS13] assert that the actual round-trip time (RTT) for in-domain and out-domain networking is 20ms and 150ms respectively; considering

an input rate of 10 Gbps, a Interest packet size of 128 bytes, and a worst-case RTT of 250ms, the maximum number of element is 2.5M at most. We can approximately consider the size of a typical PIT in the order of $O(10^6)$, as previously shown in [VPL13, DLCW12]. Our PIT module follow the usual NDN addressing scheme with flat name representation, and an exact-matching is used to perform insertion, lookups and updates. In the case of high transmission delays, or when some users are requesting non-existing content, timer support may be required to avoid PIT pollution with unnecessary items.

4.2 Design space

In this section we overview the design space explored for the PIT. Section 4.2.1 reviews the related work. We describe in Section 4.2.2 the current proposals for PIT's *placement*, i.e. "where" in a router the PIT can be implemented. Then we analyze in Section 4.2.3 a set of *data structures* which could be used for PIT's design. Timer support and loop detection are discussed in Section 4.2.4 and 4.2.5. Finally, Section 4.2.6 describes the main approaches to manage parallel access in a shared data structure.

4.2.1 Related work

Our work on Pending Interest Table could be located inside the general model of *flow-based networking* [PPK+15, MAB+08], in which a networking state is mantained in some data structure and several update or delete operations per second may occur. The data structure which maintains the flows is called a *Flow table*. In this section we show the state-of-the-art solutions addressing the same challenges of this chapter, namely the design of generic flow tables, and the specific PIT design.

Flow-based Networking OpenFlow [MAB+08] is an example of a flow-based paradigm: a flow is a tuple that is built from fields of a packet, from layer 2 (MAC) to layer 4 (TCP/UDP ports). Every time that a packet reaches an *OpenFlow switch*, the tuple[1] is exctracted from the packet, and it is matched against a flow table. If no such element is present, the packet is sent to a *Controller*, which performs a deep packet inspection, creates a flow, and sends back a *rule* representing that flow to the switch. Finally the switch updates its Flow table, and all subsequent packets that share the same tuple are forwarded according to the same rule.

[1] A typical packet's classification consists in the so called *5-tuple* extracted from the packet, which is made of IP source and destination address, TCP/UDP source and destination port and protocol number.

Counters in the Flow table may be updated for every packet. It is important to highlight that a Flow table may have several stages, i.e. a pipeline of flow tables. A match upon the i-th table may redirect to another table, or directly to the end of the pipeline. Flow table designs may be classified in *hardware* and *software* solutions.

Hardware techniques are usually preferred for the OpenFlow switches' flow tables [MAB+08]. An OpenFlow switch is equipped with a Ternary-CAM table, a HW memory that can store the bit values of 1, 0 and $*$ (don't care bit). Such an hardware solution allows fast lookup (in the order of millions of rules per second) but is very slow for the update instructions, i.e. less than a thousand update per second. Moreover a TCAM has an expensive cost per bit, and it is very power consuming.

To solve the power and cost issue, software flow tables have been introduced. OpenVSwitch (OVS, [PPK+15]) is a virtual switch framework which implements an OpenFlow switch. Authors divide the switch's flow tables in two parts: a *microflow* table and a *megaflow* table. A microflow table is implemented as a simple hash table, which exploits the cache memory of the underlying architecture. An exact match is performed on the packet tuple, and entries are extremely fine-grained (per transport connection). In case of a miss, megaflow table is accessed and a packet classification is performed, in order to do a flow update. OVS works both in user and kernel space. The kernel space application can sustain a rate of tens of thousands flow update per second, when a million flow is present in the hash table.

Both those Flow table solutions work with the Internet Protocol Suite. They allow only a poor customization scheme and it is usually limited to the type of the matching algorithm, that can be exact match, prefix match or wildcard).

PIT Designs We now focus on other PIT designs, starting with [DLCW12]. Dai et al. propose for their PIT design a Name Component Encoding method. Each component of the name is encoded, and the integer value obtained is used to build an Encoded Name Prefix Trie. This design performs well in terms of space compression (only few tens of Megabytes for a 10M dataset) and scalability (it allows longest prefix matching, with a fixed number of components), but it can sustain only slower rates in comparison to our solution (2.75 Mpps for insertion, 2.50 Mpps for deletion).

DiPIT [YMT+12] is a PIT design which adopt a *divide et impera* approach to distribute the Pending Interest Table among each interface. This design makes use of distributed Bloom filters, and it is scalable and performing on the Interest Packet processing, but show some flaws for the Data Packet processing, due to the multiple parallel lookups required to find the correct interface and complete the delivery. In their setup, with 16 Line cards, they can sustain up to

200 Mpps, which is 12.5 Mpps on each line cards. But since this design allows false-positive match, the real rate can clearly be affected (at 120 Mpps, the false positive probability ranges from 20% to 40%. Moreover, the design has been developed for 16 bytes content names.

In [YCC14] the design of a Segregated PIT is proposed. Like the previous work, authors divide the Pending Interest table among $\mathcal{N}$ interfaces, but the division of tables is more general: assuming P PITs and $\mathcal{N}$ interfaces, the PITs are distributed over the interfaces so that $\mathcal{N}/P$ interfaces share the same PIT. In a special case, $P = 1$. The design is based on the assumption of a different behavior between core and edge content routers: a segregated PIT performs an aggregation mechanism in the edge (using real content names), and no aggregation in the core (using fingerprints). This allows the deployment of fast tables on the core network, which perform an exact match against a fingerprint hash table, and another hash table on the edge, that will perform insertion, updates and deletes of real flows. This can improve scalability, but shows the drawback of Interest overheads and Fingerprint collisions. Interest overhead occurs when a flow is not recognized in the core network due to a different fingerprint, and so the Interest packet is considered different and is sent anyway to the next hop. Fingerprint collision is the dual problem, due to different content names sharing the same fingerprint. This causes a break on the current Interest packet path.

MaPIT [LLZ14] is a design of a PIT made of two components: Mapping Bloom Filters and String Hash Table. This design exploits the smaller size of the Bloom Filters, that can be stored in a fast SRAM memory, to speed-up the overall processing. Once a match is found, the String Hash table is accessed and the lookup is finished. MBF allow a tunable false positive, which is shown to be less than 0.2%. In this paper authors don't analyze the rate for insertion and deletion, but we can derive it from the building time: they claim that the time to build the MaPIT is 7488ms for 1M names: this means that, when the table is populated, a rate of $1M/7.488 = 133.547kpps$ is achieved.

4.2.2 Placement

As presented in Section 2.1, we assume a content router composed of $\mathcal{N}$ line cards which operate at a rate R, interconnected by some switch fabrics and logically separated into *input* and *output* line cards. We analyze the classical placements , which are input only, output only, input-output (cf. Section 2.1) and focus on a novel placement, called third party, that was previously introduced in [VPL13].

Input-only Originally proposed in [DLCW12] it indicates that a PIT should be placed at each input line-card. Accordingly, an Interest creates a PIT entry only in the PIT of the

line-card where it is received. When corresponding Data returns at an output line-card, it is broadcasted to all input line-cards where a PIT lookup indicates whether the Data should be further forwarded or not. This placement enables multipath (cf. Background Sec. 1.4.2) as Interest may be forwarded to multiple output interfaces, but it lacks loop detection[2] and correct Interest aggregation, as each PIT is only aware of local list of interfaces and list of nonces. Most importantly, this placement requires $\mathcal{N}$ PIT lookups in presence of returning Data, which is a serious bottleneck.

Output-only Originally proposed in [DLCW12], it indicates that the PIT should be placed at each output line-card. Accordingly, an Interest does not create a PIT entry at the input line-card where it is received, but at the output line-card where it is forwarded, selected using LPM in the FIB. This approach allows aggregating Interests received at different line-cards, but it shows limitations in case of multipath. When an Interest received at line-card i is forwarded to two output line-cards, j and k, the returning Data is forwarded twice by line-card i; in fact, the Interest creates two entries in PIT_j and PIT_k , respectively, and there is no way for line-card j and k to detect whether the Data was already received at the other line-card. Similarly, assume an Interest received at line-card i is sent to line-card j and a second Interest for the same Data is received at line-card l but sent to line-card k. In theory, the first Data received, either on line-card j or k, should satisfy both Interests; in practice, two Data are needed with this PIT's placement. Finally, the output-only placement requires a FIB lookup per Interest, even when a previous Interest for the same content was already received. Last, loops cannot be detected as each output PIT is only aware of the local list nonces.

Input-output This placement was originally discussed in [DLCW12] but dismissed in favor of the output-only placement. It indicates that the PIT should be placed both in input and output. Accordingly, an Interest creates a PIT entry both at the input line-card where it is received and at the output line-card where it should be forwarded. Compared to the output only placement, it has two benefits: no unnecessary FIB lookups and duplicated packets in presence of multipath. However, the input-output placement also suffers from the latter multipath issue as it also requires two Data in order to serve Interests received at different line-cards that were forwarded to different output line-cards. Also, loops cannot be detected for the same reasons as above. A minor problem is that a Data triggers two lookup operations: in the PIT of the line-card where it is received, and in the PIT(s) of the line-card(s) from where it was originally requested. The latter issue is discussed in [DLCW12] as the main motivation to dismiss this placement.

[2]Errors in loops may occur when a router's line card forward the Interest packet to another router's input line card: the input-only placement does not detect the loop in this scenario.

Third party The third party placement indicates that a PIT should be placed at each input line-card as in the input- only placement. However, when an Interest for a content x is received at a line-card it is "delegated" to a third party line- card, here the name. This third party line-card is selected as $j = H(x)\ mod\ \mathcal{N}$, where $\mathcal{N}$ is the number of line-cards in the router and $H(x)$ is the hash (e.g., CRC32) of the content name. Accordingly, the PIT at j aggregates all PIT entries for A independently of the input line-card where an Interest for A was received. No PIT at the output is needed; as Data is received, the output line-card identifies j by performing $H(x)\ mod\ \mathcal{N}$. This placement enables both multipath and loop detection as the third party line-card acts as an aggregation point. For example, when two Data packets are expected at two different output line-cards the first Data received is always forwarded to the third party line-card where it consumes each pending Interest. It follows that as the second Data is received and forwarded to the third party line-card, no PIT entries will be available anymore. In addition, compared to the input-output placement it only requires a single lookup per PIT's operation. The drawback of the third party placement is that it generates an additional switching operation for both Interest and Data. Such increase in switching operations can be absorbed by additional switch fabrics as commonly done in commercial routers [VPL13].

4.2.3 Data structure

In order to have a minimum memory usage, the data structure used for PIT should have a small memory footprint. PIT is accessed for every received packet, both Interest and Data, and several lookup, insert, or delete operations per time unit may be required. In the literature, three data structures have been proposed for the implementation of the PIT: counting Bloom filter [LBWJ12, YMT+12], hash-table [KMV10, YSC12, PV11], and name prefix trie [DLCW12]. We assume that a PIT entry, without regards to any kind of implementation, contains the tuple:

$$< content\_name, list\_interfaces, list\_nonces, expiration > \tag{4.1}$$

Counting Bloom filter (CBF) A CBF is a data structure for membership queries with no false negative probability and tunable false positive probability. Compared to a classic Bloom filter, CBF enables deletion using a counter per bit. In [LBWJ12, YMT+12], the authors propose to use a CBF to implement the PIT. A CBF-based PIT only stores a footprint of each PIT's entry, i.e., available or not, which realizes great compression. The drawback is the presence of false positives that generate wasted Data transmissions. Also, a CBF-based PIT can only be coupled with the input-only placement since the compression of its entries loses the information contained in list interfaces which requires to lookup PITs at each input line-card in order to determine where a Data should be forwarded. Finally, a CBF-based PIT cannot

detect loops and support timers, as nonce values and timestamps are lost in the compression as well. The memory footprint of a Bloom filter, given a fixed false positive probability f_p, is $S = \frac{-kn}{log(1-f_p^{\frac{1}{k}})}$ where k is the number of hash functions, and f_p is the false positive probability. However, a CBF requires $c \cdot S$ memory, where c denotes the size of a counter.

Hash-table It is a data structure that maps keys to values. We explored the possible designs of hash tables in Chapter 3 for the FIB (cf. Section 3.2.3, page 51). However it is worthy to explore the design space of hash-tables that could meet the different requirements of the PIT module. In [YSC12, PV11], the authors suggest to implement the PIT using a hash-table where a content name is used as key and its corresponding PIT's entry is used as a value. Compared to the CBF-PIT, a PIT based on a hash-table can be coupled with all placements, and it can detect loops (if the PIT placement supports it as well) and support timers. These features come at the expense of a larger memory footprint compared to CBF. In theory, a PIT based on a hash-table can perform all operations with a single memory access. In practice, multiple accesses are needed in presence of collisions, i.e., when multiple keys map to the same bucket. We analyze hash-table design for the PIT with the same approach of Chapter 3. In the following, we always assume a load factor $\alpha = 1$, that is the number β of buckets available in the table matches the number n of items to be stored. This analysis is derived from the work of Vargese *et al* at [KMV10].

A classic hash-table uses chaining, i.e., a list per bucket, to handle collisions. Chaining guarantees that PIT operations are accomplished in $2+\alpha/2$ memory accesses on average, where $\alpha = \frac{n}{\beta}$ and β refers to the number of buckets. However, when collisions happen, up to $O\left(\frac{log(n)}{log(log(n))}\right)$ accesses (under the assumption $n = \beta$) are needed, which can severely hurt the required determinism. The acronym to indicate a simple linear hash table is LHT. Several approaches exist to improve upon the classic hash-table with chaining [KMV10]. Multiple choice hash-tables, as d-left hashing, are data structures where d hash functions ($d \geq 2$) are used: in the case of d-left hash table (DT) each entry is hashed d times and added to the less loaded bucket among the d identified. This strategy trades increased complexity and average access time, computation of d hashing functions and d probes to the data structure, with lower collision probability, which in turn reduces the number of memory accesses in the worst case, e.g., $O\left(\frac{log(log(n))}{d\phi_d}\right)$ (assuming $n = \beta$) where ϕ_d is the asymptotic growth rate of the d-th order Fibonacci numbers (e.g. the dominant root of $x^d = 1 + x + ... + x^{d-1}$). Open-addressed hash-tables are another solution where every bucket stores a fixed number of items; the size of a bucket is limited by the amount of data that can be read with a single access to the memory. Bucket overflow is managed by chaining with linked lists, but this is expected to be rare if a bucket is large enough with respect to an item. It follows that even in presence of collisions a single memory access

is enough in most cases. The drawback is a larger memory footprint compared to the previous hash-tables. Also, bucket overflow is frequent if a bucket can only contain a limited number items. Hash-tables with index are often used to solve this last issue [FAK13]. An index consists of multiple buckets, each bucket has a fixed number of index tuples <tag, offset>. A tag is the indexed item's hash value, and the offset is used to address the actual item that is stored in a separate memory location. When open-addressing is used, no pointers are usually stored inside the buckets, since a set of slots has been previously allocated. Pre-allocation of memory space allows lazy mechanisms for element deletion: when a stored element has to be removed, a tag can be set to indicate that the location may be reusable for other insertions. Setting a tag is a simple operation that is less time consuming than freeing memory and updating pointers.

Name prefix trie It is an ordered tree used to store/retrieve values associated to "components", set of characters separated by a delimiter; for example, `INRIA` is a component in e.g., `/INRIA/THESIS/MyThesis.pdf/chunk0` and the delimiter is `/`. The name prefix trie supports LPM, and exact matching as a subset of it. The Encoded Name Prefix Trie (ENPT) [DLCW12] reduces the memory footprint of a name prefix trie by encoding each component to a 32-bits integer called "code". The drawback is that this compression requires to introduce a hash-table to map codes to components. The ENPT-based PIT described in [DLCW12] does not specify any mechanism to detect loops and remove PIT entries with expired timers; however both operations can be accomplished assuming the usage of the PIT tuple 4.1. To do so, we simply have to add to each PIT's entry the code associated to the content name. Similarly, the lazy deletion mechanism discussed above can be used to remove entries when needed. In a ENPT-based PIT, each operation starts at the root of the trie and proceeds iteratively along the tree until a leaf node is reached or it is not possible to further proceed: it follows each PIT operation requires a number of accesses to memory that is linear with the number of components in a content name. Recall that a ENPT-based PIT also require an external hash-table to store PIT tuples: it follows that the memory footprint of a ENPT-based PIT is the size of the ENPT plus this hash-table. Since no details is further provided on which hash-table should be used in [DLCW12], we assume either LHT or DT for the reasons discussed above. Finally, two additional accesses to memory are required to retrieve/update/remove an element from the hash-table once the node in the ENPT is found.

4.2.4 Timer support

Timers are used to invalidate pending requests which have not been satisfied after a given amount of time. More specifically, a timer is associated to a every active PIT entry, and, once the timer expires, the PIT entry is invalidated. Proper timer tuning is crucial for the

performance of an NDN network but it is out of the scope of the thesis (for more details, see for example [CGM12]).

Timers can be handled in an active or lazy fashion depending on whether timer expiration is detected and processed immediately or after a certain amount of time respectively. Active timer management requires to constantly verify the status of all timers and invalidate a given PIT entry as soon as its timer expires. The active approach guarantees deterministic PIT operation execution at the cost of processing resources constantly devoted to timer management.

Lazy timer management delays timer verification to periodic checks or to the moment when an entry is accessed for processing an Interest or Data packet. Once timer expiration is detected the entry is then invalidated before any further operation. The lazy approach does not constantly consume processing resources but may hurt PIT operation determinism and delay critical actions that could be associated to a timer expiration (e.g. request retransmission).

4.2.5 Loop detection

As previously said, a nonce is a random number generated by a user who wants to retrieve a content at a certain time. It is used to detect loops of Interest packets that otherwise would be forwarded continuosly. Each unique nonce can be computed as a pseudo-random value initialized with the tuple seed: SEED= $\{U_i, I_{C_j}, T_k\}$. The seed so-defined takes into account that user i is producing an Interest for the chunk j at a certain time k, and so it unequivocally identifies a single user interaction requesting a specific content chunk. Each PIT entry should maintain a list of nonces which have been seen for the content requested.

A not negligible amount of processing capacity is required to compare the nonce carried in the Interest packet with all the nonces in the PIT entry. We can optimize the time spent to traverse the list of the nonces by using *mini Bloom filter* rather than a list. In this scenario, the 64-bit nonce carried in the Interest packet is split to form four 16-bit hash values. In the corresponding PIT entry, the 64-bit nonce field is accessed as a bloom filter, and 4 bits are checked (cf. Section 3.2.3). If there is no match, then the nonce should be added. We remark that Bloom filters have no false negative probability, and a tunable false positive probability.

In particular, if we consider the false positive formula: $f_p = (1 - e^{-\frac{nk}{m}})^k$ when $m = 64, k = 4, n = 10$, the false positive is less than 5% [Blo70]. When a false positive occurs, the nonce is erroneously thought to be already present in the PIT, and therefore the interface which the Interest arrived from is not recorded. This means that when a Data packet related to that pending request reaches the PIT, it is not forwarded to that interface. This behavior affects only multicast applications, and it can be mitigated by updating the interface bitmask even

though the nonce bloom filter gives a positive response. The cost is in term of overhead in the data transmission, because a Data packet can be forwarded to some interface which did not requested a content, and the update of the bitmask was only an Interest packet which looped (we choose this approach for our design, cf. Section 4.3.5 for more details about the detection of loops with Bloom filters).

4.2.6 Parallel access

We identify three main approaches for concurrent PIT access, which vary in the level of parallelization they can exploit.

Locked In a locked access approach only one core at a time can perform insert, remove and lookup operations. Despite the ease of the implementation, its simplicity translates to a worst case because it does not exploit parallelism even in the presence of many idle cores, resulting in a one-at-a-time behavior for each packet.

Load balancing A classical approach to allow many cores to process data in parallel is to provide memory access by using a load balancing (LB) technique. The load balancing approach consists in giving all the cores a separate and private subset of the memory that can be used for lockless instructions. The LB can virtually exploit full parallelism (under the hypothesis of perfect distribution of packets, that grants a fair load balance among the cores): since one of our main goals is high-speed computation, we decide to adopt this approach for our main PIT table.

Reuse Finally, a reuse approach refers to a method in which every core access a private buffer of elements that can be used to locally store and delete entries. Since local buffers have a non-shared visibility among multiple cores, all local operations can be considered lockless (similar to the LB). When the system reaches saturation (i.e. one local buffer is full), memory resources should be reallocated, and the overflowing local buffer must be expanded. The expansion operation accesses a shared memory area, and therefore it is locked. When a lookup or an update occurs, the core should decide whether a matching PIT entry belongs to its private buffer in order to perform a lockless instruction; otherwise it should lock the corresponding memory area to access another core's local buffer. This intermediate step affects only the update and delete instructions, while the lookup is always lockless. This scheme represents a trade-off between the previous cases.

4.3 PIT: design and implementation

Following the insights of the previous section, we now describe the design of our Pending Interest Table and its integration in our content router Caesar. First, we review our content router architecture and discuss the PIT placement. Then, we present the PIT data structure and its main operations.

4.3.1 PIT placement and packet walktrough

We integrate our PIT design in Caesar, with both input-output and third-party PIT placement; as described in Section 4.2 these two placements enable all NDN features. The packet workflow is as follow.

- *Packet Input:*
  The packet is received on an input line card, and a hardware load balancer dispatches it to one of the available cores of our architecture for processing. The load balancing can be either uniform random across cores or based on the hash value of the full content name carried by the packet. The former guarantees a uniform distribution of load across cores, but it requires mechanisms to deal with concurrent operations on the PIT performed by different cores. The latter may result in unbalanced load over cores in presence of non uniform workload, but it allows concurrent PIT operations as every core works on an isolated part of the PIT.
- *Packet processing*:
  - **Input/Output placement:** In case of an Interest packet, PIT *insert* or *update* operations are performed on the input line card, and the packet is transferred to the output line card towards the next hop via the switch. In case of a Data packet, PIT *remove* operation is performed and the packet is then transferred to line cards that have requested the corresponding data as indicated in the PIT entry.
  - **Third party placement:** The target line card i is computed by using the hash value H of the full content name, as $i = H \bmod \mathcal{N}_L$. The packet is then transferred to line card i for PIT processing via the switch. PIT *insert*, *update* or *remove* operations are performed and the packet is then transferred to the output line cards via the switch.
- *Packet output:*
  Once received by the backplane interface of the output line card, the packet is assigned

to a Caesar's core and the packet is sent to the interface for transmission. In case of input/output PIT placement, PIT operations are repeated on the output line card before the packet is transmitted.

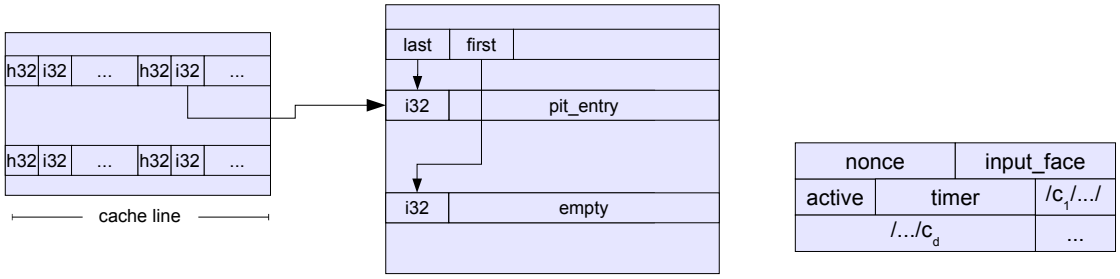
4.3.2 Data structure

Among the most suitable data structures identified in Section 4.2.3, we base our PIT design on a hash-table with index approach for two main reasons. First, in our architecture the amount of memory that can be read with a single memory access is 128 Bytes, which corresponds to a cache line. As a PIT entry is about 100 Bytes large, a simple open-addressed approach would result in frequent bucket overflows. Second, with a multiple-choice approach the complexity overhead of multiple hash value computations would be higher than the complexity overhead required to solve the collisions generated with a single hash value.

Our design is detailed in Figure 4.1a. It consists of an entry table, that manages and stores PIT entries, and a hash index, to quickly find entries. The *entry table* stores fixed-size PIT entries, and is organized as a circular append-only array where every entry can be addressed with his position in the array. Values associated to a novel content name are simply inserted in the PIT entry right after the last used one, overwriting existing values. We dimension the entry table to handle worst case scenarios, where $n = N_{MAX} = \lambda_{in} T_{MAX}$ (cf. Section 4.1) so that overwritten values are necessarily obsolete (i.e. the PIT entry is expired). The *index* consists of β buckets of 128 Bytes each (i.e. a cache line). Each bucket is composed of s slots, consisting of the index tuples $< H(content\_name), PIT\_entry\_position >$: the first field is a the 32-bit CRC value computed over the full content name, while the second field is 32-bit value that identifies the PIT entry position on the entry table. We choose to have a load factor $\alpha = 1$, and therefore $\beta = n = 1M$

The value s identifies each bucket's size of the index table. The maximum number of slots per bucket represents a worst case for the elements lookup: when the bucket is full, and the desired element is at the end of the bucket, at least s trials are needed. Our design choice is to set $s = 13$, even though it does not cover the whole cache line, and padding the rest of the bucket with dummy bits.

The PIT entry is detailed in Figure 4.1b, and consists of the fields described in Section 4.2. The hash value of the content name is omitted as it is already present in the index; the content name itself is stored in a separated memory area to handle variable size names and a 64-bit pointer to the name is stored in the PIT entry. A bloom filter of 128-bits is used to keep track of the nonces for loop detection (cf. Section 4.2). 128-bits are reserved for timer management,



(a) *PIT Hash table* **(b)** *PIT entry*

Figure 4.1: *PIT table and the single PIT entry.*

while a 128-bit bitmask is used to identify the interfaces from which the Interest packet has been received. It follows, the size of a PIT entry is 58 Bytes (aligned to 64 Bytes to fit half cache line), plus 42 Bytes required to store a content name on average.

Parallel access In Section 4.2.6 we described three different approaches to manage the parallel access to the main memory. Our PIT is designed to exploit parallel processing of multiple cores that share the same DRAM memory area. In our design, the PIT hash table is therefore split in many subtables, and every core can perform insert, lookup and updates in a lockless way. The procedure to achieve the lockless operations is the following: each packet is first hashed, in order to find which core is responsible for that content name, and it is forwarded to the corresponding core to be processed. Once the packet is dispatched, the core can perform the usual NDN operations without concurrency issues. This mechanisms works both for Interest and Data packets.

We chose the LB approach as our default design to exploit Caesar's parallel multicore architecture, and because of the simplicity of deployment. Under the hypothesis of a fair packet distribution among the cores (thanks to a *perfect hashing*) it may result in a performance speed-up w.r.t. the other approaches. However, we evaluate in Section 4.4 the LB PIT, together with the Reuse and Locked approach.

4.3.3 PIT operations

As described in Section 4.1 three main operations are performed on the PIT: *insert* or *update* when an Interest packet is received, and *remove* when a Data packet is received or a timer expires. Upon reception of an Interest or Data packet a preliminary *lookup* operation is required to verify the existence and retrieve a given PIT entry. In the following we describe how those four main operations are performed on the data structure described above.

Lookup The lookup operation works as follow. First, we generate a hash value H from full content name extracted from the Interest or Data packet, and identify one among the β buckets performing a modulo instruction: the index $i = H \ mod \ \beta$ is then found in the index table. Bucket i is loaded in the main memory, and H is compared with all the hash values stored in the bucket items until a match is found. If a match exists, the content name is compared against the name stored in the PIT entry indexed by the matched index item. If the two names correspond, a PIT entry for the incoming content name exists; PIT processing continues with an update or remove operation for an incoming Interest or Data packet respectively. Otherwise, the lookup in the index table continues: this event is due to hash collision but anyway it is expected to be rare for well designed and dimensioned hash functions.

If there is no match at the end of this process, then a PIT entry for the incoming content name does not exist. In case of an incoming Interest packet the PIT processing continues with an insert operation, while in case of an incoming Data packet the packet is dropped as it has not been requested.

Insert The insert operation consists of the following steps: i) add the required information to the first available PIT entry in the entry table; ii) update the hash index. The first operation is performed by filling the fields of the PIT entry right after the last used one. The second operation consists on writing some data to the bucket $i = H \ mod \ \beta$ (previously selected by the lookup operation); we insert the hash value H and the position of the PIT entry in the entry table in the first available item of such bucket. Chaining with linked list is used if all items are busy, but this is expected to be rare if the number of buckets is large enough.

Update The update operation consists in updating the information stored in the PIT entry identified by the lookup operation. First, the incoming nonce is checked against nonces stored in the PIT entry. Then, if the nonce for the considered content name has not been received before, the nonce is added to the nonce Bloom filter field. Also, the interface from which the interest has been received is added to the list of interfaces.

Remove The remove operation is performed when a Data packet is received, or as a consequence of a timer expiration. In the both cases the active field of the index tuple identified during the lookup process or specified in the timer entry is set to $FALSE$. In the former case the timer expiration is also descheduled.

4.3.4 Timer support

Our timer management scheme follows a mixed active-lazy approach (cf. Section 4.2.4). Specifically, we leverage hardware functions available in Caesarto schedule PIT timer expiration. Then, expired timers are handled with low priority by Caesar's hardware cores, i.e. a core handles a timer only if there are not packets to be processed. Finally, before any PIT operations described in the previous section is executed a given core checks if the timer associated to the concerned entry is expired.

Our design does not require any additional processing resources to monitor and detect timer expiration. It promptly reacts to timer expiration in most cases, as 100% core load is a rare event and happens for extremely short period of time. Finally, it does not impact PIT operation determinism because: i) timer expirations are handled separately from regular PIT operations in most cases; ii) if a timer expiration is detected once a PIT entry is accessed for a regular operation, the invalidation of a given entry only requires to set the *active* field of a the index entry to FALSE.

When a PIT entry is used, a timer value is stored together with other information. A timer is represented by a 32-bit integer value, which refers to the clock cycle of the insert operation. An update instruction causes an update of the timer value, while a delete instruction does not affect the value stored in the timer; in fact, when a timer expires, the active field is set to zero and there is no need to overwrite the previous value of the timer. When a lookup occurs and a PIT entry is found, the timer value is compared to the current clock cycle: if the difference is higher than a tunable threshold, the PIT entry is considered expired, and it is afterwards invalidated. We fixed a threshold to span a lifetime of $500ms$ at most. This mechanism enables lazy deletion of expired entries, both for the Linear and the Index Hash table: in the first case, an entry is checked while pointers of the linked entries are traversed, while in the second one, timers are compared during the traversal of all the slots inside the current bucket.

To speedup the overall processing, the 32-bit timer is stored in the index table, and therefore the index tuple becomes the following: $< Timer, H(content\_name), PIT\_entry\_position >$. When a lookup occurs and the bucket is chosen, then all the timers in the bucket can be checked and invalidated if needed. The presence of the 32-bit timer in the index table reduces the number of slots available in the bucket. We choose to fix $s = 6$ when timers are enabled.

4.3.5 Loop detection with Bloom filter

Bloom filters grant membership queries with no false negative answers, and a tunable false positive probability. Since we adopt a BF approach to store the nonces indicating looping

or duplicate packets, no loops are created (no false negatives) but some non-looping Interest packet may be erroneously considered as a duplicate (with a probability $f_p = 0.05$, derived in Section 4.2.5).

Such an inconsistency affects our performance in case of multicast applications, i.e. lots of users requesting the same content chunk in a small time frame. We believe that this approach is naturally mitigated by the aggregation feature of the PIT, and can be furtherly mitigated bypassing the BF for popular content requests.

Aggregation and looping Interest packets We now consider an Interest packet which is incorrectly considered a duplicate. This packet will be discarded, and the corresponding interface I will not be added in the PIT. However, for popular content, it is likely that I is already present in the PIT entry's interface list due to some other content request. In this case, even for the non-looping Interest packet, a corresponding Data packet is still created and sent back to I. Thus, only a fraction of the false positive can be effectively considered as the probability that a request is unserved. In case of unserved requests, retransmission may be needed, and a traffic overhead can be present.

Ignoring the BF nonce for popular content In Background Section 1.4.1.3, Figure 1.7 we described that a Processing module may be present in a content router. This can be used to monitor the popularity of requested contents, by counting the name occurrences in the PIT. It is possible to detect the requests which are more popular (which are the ones that may cause a BF nonce failure) and send Interest packets even in case of a false positive occurrence. This may cause two problems: first, a general traffic increment; second, the forwarding of Interest packets which are looping in the network.

Store the nonce list in the DRAM Finally, we can use the Bloom filter nonces together with a nonce list stored the DRAM. In this way, a new nonce is added both in the nonce BF as a "fingerprint", and in the nonce list with its complete value. This approach relies on the benefit of the BFs, which reduce the number of useless accesses to the separate data structure, together with the possibility to handle the false positives thanks to the access to the actual nonce list. However, the nonce list is a very dynamic data structure (i.e. very frequent modification, due to the incoming new nonces) and can dramatically reduce the overall PIT performance due to atomic accesses and updates.

We decide to adopt the approach of the BF nonces and handle the possible false positives (creating unserved requests) with retransmissions, at a cost of an Interest traffic overhead.

4.4 Evaluation

In this section we evaluate our Pending Interest Table. We first describe our methodology in Section 4.4.1 and then we present our experimental results from Section 4.4.2 to 4.4.4.

4.4.1 Experimental setting

For the experimental evaluation of our PIT we use the typical settings showed in Section 2.4.2.

Our Pending Interest Table is statically allocated at the start-up of the content router, and may contain a tunable number of buckets. The maximum capacity of a PIT should be sufficient to save the state of all pending requests: this size has been shown to be of the order of 10^6 elements [DLCW12] (cf. Section 4.4.1.1). We compare our PIT design with a linear hash table implementation. We also show the difference between the locking mechanisms described in Section 4.3.2.

We assume as workload the reference workload shown in Section 2.4.2, and we use it as input for our commercial traffic generator to generate Interest and Data packets.

We define *traffic load* as the ratio between Interest packets and the total number of packets. We identify two main conditions for the traffic load. We call *flow-balanced* scenario when all the Interest packets are satisfied by Data Packets; in this scenario, the load is 50% (i.e. 50% of all packets are Interest packets). We can vary the load conditions by varying the number of Interest packets satisfied by the corresponding Data packets between 50% and 100% (i.e. all packets are Interest packets). This is called the *Interest-only* scenario.

A flow-balanced scenario represents an average condition, in which all pending Interests are satisfied by a corresponding Data packet. If the network is overloaded, packet losses may occur and not all Interests can be satisfied. In this condition the number of PIT elements increases, and we can observe a loss in performance due to the slower insert/update operations.

Our experiments consist of the following steps: first, traffic with desired characteristics is originated at the traffic generator and transmitted to Caesar's line cards; then Interest and Data packets are processed by line cards and content names are extracted; finally, PIT is accessed both for Interest and Data packets. According to the match result of the insert/lookup operations, forwarding decisions are taken and packets are sent back to the generator. We then measure the typical characteristic values of forwarding rate and packet latency.

4.4.1.1 PIT dimensioning

Before to analyze the memory footprint of our PIT table in Section 4.4.2, we analyze the procedure for a correct PIT dimensioning. The reason behind the choice of a particular PIT size deserves some considerations about the typical memory consumption of the PIT table. We detail in this section the analysis of the PIT's large state showed in Section 4.1 at page 4.1. We consider two variables: the insertion rate and the deletion rate of PIT entries. Let λ_{INS} is the insertion rate for a specific line card: its value is not necessarily equivalent to the interface's maximum rate, because it takes into account only the new incoming Interests which are not already stored in the PIT. We can although consider the worst case for the insertion process, that is when all incoming requests refer to contents not yet requested. In this case, $\lambda_{INS} = \lambda_{INTERFACE}$.

Let now μ be the deletion rate. We remind that a PIT entry is deleted from the PIT as soon as either a corresponding Data packet is received, or the timer related to the item expires. We can assert that $\mu = \lambda_{DATA} + \lambda_{EXPIRATION}$. For the deletion process, the worst case occurs when all the requests are not matched by an incoming content data (and therefore $\lambda_{DATA} = 0$). In this scenario, all the elements are kept in the table until timers expire, and the deletion rate is equivalent to the timer expiration rate.

This simple analysis can be used to dimension our PIT table in the worst case, that is when $\lambda_{INS} = \lambda_{INTERFACE}$ and $\mu = \lambda_{EXPIRATION}$. Given a 10Gbps interface, assuming Interest packet of 120 bytes, and an expiration time of 500ms, the maximum number of elements in the PIT table is bounded by $\frac{10 \cdot 10^9 \ bit/s}{120 \cdot 8 \ bit} \cdot 0.5s \approx 5 \cdot 10^6$ elements.

It is reasonable to tune our table to contain 1 million elements because the worst case is a rare event into common conditions. In fact, PIT allows to aggregate the content requests for the same content name (resulting in $\lambda_{INS} < \lambda_{INTERFACE}$), and if the network is not under attack or in a congestion situation, requests will be eventually matched by corresponding Data packets (therefore $\lambda_{DATA} \neq 0$).

4.4.1.2 PIT bucket overflow

We observe Figure 4.2, showing the number of elements per bucket when the number of PIT buckets β is 1M for a workload of $n = 1$M elements. When timers are not enabled, the bucket size is $s = 13$, and therefore there is never a bucket overflow. However, when timers are enabled the bucket size is reduced to $s = 6$ slots: we derived that about 100 buckets over 1M are overloaded. We evaluate that, assuming no timer expires in one time slot, the probability of overflow in our dataset is $\mathcal{P}(overflow) = 1.2 \cdot 10^{-4}$.

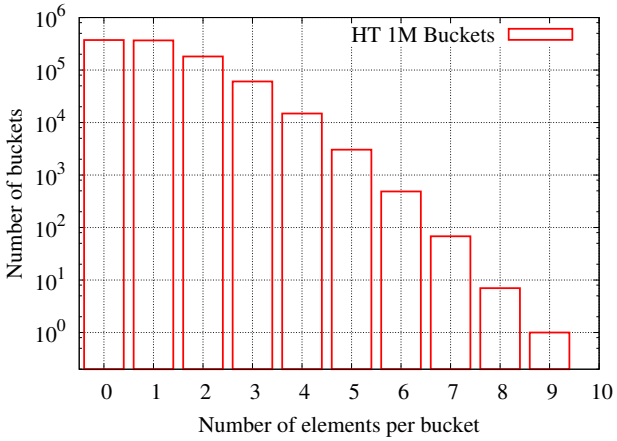


Figure 4.2: *Number of prefixes per bucket in the PIT. A bucket overflow occurs only when timers are enabled.*

Hash Table	Overall Memory	Index	Entries
Linear	152 MB	NA	152 (variable)
Index	232 MB	123 MB	108 MB

Table 4.1: *Memory usage of Linear and Index Hash Table*

Overflows are managed by overwriting the first slot of the overflown bucket. We give two main reasons to explain how the error rate of this approach can be mitigated. First, when timers are enabled it is more likely that some bucket slots are freed due to the timer expiration: in fact the calculated $\mathcal{P}(overflow)$ is an upper bound, and the real percentage of items which will be overwritten is generally lower. Second, when one item overwrites another slot in the same bucket, the latter is always older than the current one.

Additionally, buckets overflow can be mitigated by reducing the value of the PIT entries' expiration time.

4.4.2 Memory footprint

We begin our analysis by considering the memory footprint of our PIT. The memory footprint is calculated as the size of the memory allocated for the Pending Interest Table which is stored in the DRAM memory at the startup of our experiments.

In Figure 4.3 we show a comparison between the memory occupied by both Index and Linear PIT as a function of the number of buckets. The size of the Index HT is made of two components: the preallocation of all the indexes, and the memory for the actual entries. Both components of course grow with the number of buckets, and the overall PIT size grows following the sum of the two parts. LHT makes use of pointers instead of indexes, and therefore the

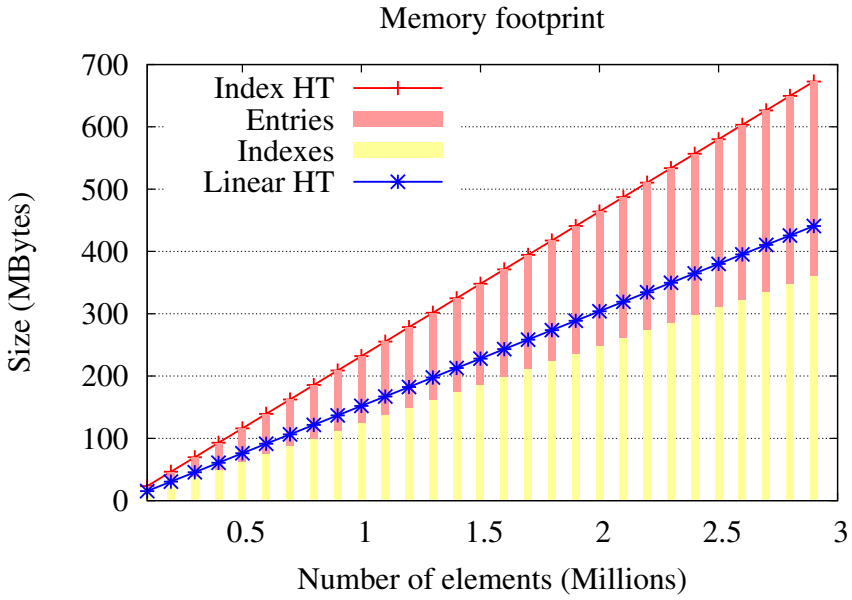


Figure 4.3: *Memory footprint for Linear and Index PIT as a function of the number of buckets*

PIT entries and the corresponding pointers used to manage collisions occupy the full amount of memory space of the LHT, being physically located in the same memory area.

We observe that the Index Hash table uses more memory than the LHT. This can be explained considering that the Index HT is overdimensioned with respect to the incoming content names. As described in Section 4.3.2, each bucket may potentially contains up to $s = 13$ slots: therefore, if needed, the whole table can sustain a number of elements which is about ten times the number of buckets, at the price of increasing the number of preallocated PIT entries. On the contrary, the LHT requires less memory than the Index HT, but considering a population of 1M elements, LHT is still comparable to the Index table. We remind that the size of pointers stored for every item in LHT is 64 bit, while in the Index Hash table all items are indexed with a 32-bit index value. Since each entry is linked to the following and the previous on, the size of pointers is not negligible in the memory footprint.

A summary of the results is shown in Table 4.1, where we choose the reference bucket's number of 1 million. The Linear Hash table implementation needs 152 MB of memory. The amount of space used by the Index hash table implementation is 232 MB, which consists of 123 MB used by the actual indexes and 108 MB used by the complete PIT entries.

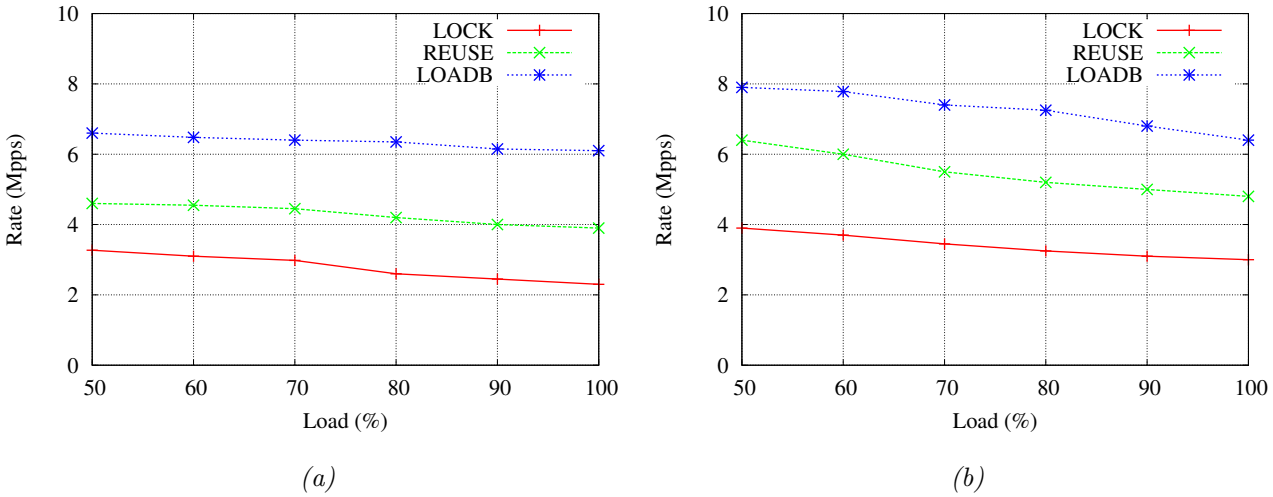


Figure 4.4: *Throughput of the Linear (a) and Index Hash Table (b) as a function of the traffic load as defined in Section 4.4.1 at page 95.*

4.4.3 Throughput without timer

Figure 4.4 (*a*) and (*b*) reports the plot of the throughput, measured in millions packets per second, as a function of the traffic load in the network, as defined in Section 4.4.1.

For both Linear and Index Hash table, the Load balancing approach (the default for our design) takes advantages of the lockless operations, and so it performs better than other schemes for concurrent access. For instance, in a flow-balanced scenario LHT and Index HT reach a throughput of 6.2 Mpps and 7.9 Mpps respectively. The throughput decreases as the load increases: in the Interest-only scenario (i.e. traffic load is 100%), the PIT throughput reaches 6.1 Mpps and 6.4 Mpps for the Linear and the Open Hash table respectively.

The plot highlights that the LHT implementation provides a throughput which is almost constant, independently from the load. On the contrary, the Index HT is almost 2 Mpps slower when the flow is made of Interest packets only, with respect to the flow-balanced scenario. We can explain this behavior by considering the bottleneck of the PIT in the two opposite scenarios. In a flow-balanced scenario lots of insert/deletions should be performed: this keep the usage of PIT low, and several buckets are free to use. The Interest-only scenario, on the contrary, allows only insert and update operations (up to the saturation of the buffer of buckets), and therefore the hash table is always full. When the PIT is full, and timers are not active, every incoming item is checked to detect if any existing element may be updated, and several comparison (and possibly updates) are performed.

The LHT is slower in the flow-balanced scenario, due to PIT entries being continuously deleted by a corresponding Data packet, or created by an incoming Interest. Pointers management

slow down the LHT performance. Index deletion, insertion and update are faster thanks to the pre-allocation of the PIT entries. In fact, we remark that all deletions translate in the overwriting of a variable in a PIT entry.
In the Interest only scenario, performance are similar between Index and Linear Hash table. In this scenario, PIT bottleneck is update-bounded, because the majority of the operations are updates. LHT is not strongly affected by this situation because its bottleneck is already present in the pointer traversal operations. Index HT's drop in performance is caused by the fact that when the hash table is full, all the slot of a bucket must be traversed to detect whether a slot is available or not. This traversal is not present in the flow balance scenario. Since both LHT and Index HT are affected by the complete traversal of a bucket in the Interest-only scenario, we expect to observe similar performance between the two approaches: experiments confirm this equivalence between the two implementation.

The Load balancing implementation might be affected by the type of the traffic injected, since some adversarial traffic pattern may overload some core with respect to the others, causing a loss in performance. The case of such a pattern is very unlikely, and in the literature several works make use of a LB approach: these are the explanation why we choose this approach as our default design. When it is necessary to avoid cores overload, a tradeoff can be made and other approaches can be adopted. In our experiments we show as well the performance comparison of the Reuse and Locked HT. The plots of these other approaches report a behavior which is similar to the LB design for both Linear and Index Hash table, with the former being quite constant as the traffic load varies, and the latter showing better performance when under a flow-balanced scenario. The overall summary of this evaluation is that our design for the PIT module is the one that performs the best in the majority of the scenarios.

4.4.4 Throughput with timer

In Section 4.3.4 we described the design of the timer management for the Pending Interest Table. Thanks to our mixed approach using both lazy and active deletion, timers activation may introduce some processing overhead due to the 32-bit write or comparisons. This is the main difference with respect to the previous experiment. Our approach grants that timers instructions occur in two scenarios: first, for an incoming Interest packet corresponding to a PIT entry, to detect if the existing entry is expired or it can be updated, and second, for the other PIT entries encountered during the bucket traversal of the lookup process, to check if some item can be lazily invalidated. As shown in Section 4.3.2, a bucket is entirely in a cache line: this enables fast operations on all the bucket entry.

Experimental results show that no significative difference exists between enabling or disabling

timers. The timer expiration occurs even in a 100% Interest scenario, because the incoming Interest not only triggers the timer check for an existing PIT entry, but also for all the entries sharing the same original bucket. As a result, PIT is never overloaded even in congestion conditions: thanks to the timer expirations, some slots are freed on Interest lookup,and new requests are allowed to be inserted and removed.

A second explanation to this behavior can be given considering the bucket size s. When timers are enabled, less slots are used for each bucket, and so the worst-case lookup, in which s trials are required, consists of at most $s = 6$ trials, which is half of the worst-case when timers are not enabled. Write instructions, which could affect the overall throughput, are mitigated by the less number of trials per bucket in the worst case, and by the fact that simple 32-bit comparison or item invalidations always occur in the same cache line. One of the main findings of Chapter 3 is that exploiting memory cache lines of the underlying architecture provides a significant speedup. This appears as another evidence of the cache-line performance gains.

After this analysis, we chose to focus on the LB design, with timers enabled. This choice relies on the fact that the experimental results about PIT's throughput show better performance both in a flow-balanced scenario (which represents an average case) and in a Interest-only scenario (worst case). A Reuse approach is more reliable than the Load balancing approach because it is not affected by some particular traffic pattern, but the observations of the Section 4.4.3 represent valid reasons to focus on the performance advantages of LB rather than the tradeoff of the reuse.

4.5 Conclusion

In this chapter we focused on the design and implementation of the Pending Interest Table (PIT), a module of our content router Caesar, which is able to maintain a soft-state of the pending requests. PIT allows aggregation of the same content requests as well as symmetric routing, two core features of NDN. The PIT prevents the creation of loops of packets and enables a native multicasting at network layer.

We make the following contributions. First, we investigated the spectrum of candidate designs for PIT, focusing on its placement with in a content router, on the best-fitting data structures and the necessary features to enable a full NDN processing. Then, we showed our design for a PIT, and proposed a module which can be easily integrated in Caesar, our content router. We evaluated each design with respect to PIT's requirements on our prototype, performing extensive experiments by injecting traffic with a commercial 10Gbps traffic generator.

Experiments showed that we can sustain a rate of several Gbps even in the challenging situation of several insert/update/remove operations occurring at a per-packet granularity. Caesar's modular design is therefore proven to be flexible and extensible.

Table of symbols, part I

Content router, dataset, chassis	
LC_i	Line card number i, for $i \in [0, \ldots, \mathcal{N} - 1]$
LT	Line card table, used in the Distributed Forwarding extension
n	Number of elements in the dataset
$\mathcal{N}$	Number of line cards in a router
R	Rate of a single line card
Content names and prefixes	
d	Number of components of a prefix
p	A generic prefix
p_i	A generic prefix of i components, that is $/c_1/c_2/\ldots/c_i$.
t	Distance between prefixes, measured in number of components
x	A generic content name, that is a prefix followed by a chunk identifier.
Hash table design and evaluation parameters	
α	Load factor of a hash table
β	Number of buckets in the hash table
H, h	A generic hash function. (CRC32 is our reference $h(\cdot)$)
ρ	Fraction of delegated packets in the Distributed forwarding
s	Number of slots in a bucket
Bloom filters and PBF	
b	Number of blocks in the prefix Bloom filter
c	Size of a counter in a counting Bloom filter
f	False-positive rate for a single lookup in the PBF.
f_p	False-positive probability of a generic Bloom filter
k	Number of hash functions used for the hash calculations
M	Size of the Bloom filter, measured in bits
m	Size of the prefix Bloom filter block, measured in bits
n_{ij}	Number of elements of j components inserted in the i-th block of the PBF
n_i	Number of element inserted in the block i of the prefix Bloom filter
$\mathcal{P}(b_i)$	Probability of choosing the block i in the prefix Bloom filter
TV	Threshold value for the block expansion in the PBF
w	Width of the expansion bits in the preamble of a PBF block

Part II

Network Verification

Chapter 5

Introduction to the Second Part

Network incidents are not rare events: on the contrary they occur frequently and the damage can vary with the size of the network and the typology of the incident [GJN11]. Some of the network errors may be due to network bugs, misconfiguration or failures of some nodes. The interest for network problem diagnosis has recently grown after the advent of SDN [PKV+13, KDA12, MRF+13, KRW13]. Network diagnostic can prevent and/or detect the manifestation of malicious events, but it is often a time-consuming operation, especially with the complexity and unpredictability of today's networks. In fact, a network may consist of several elements (hundreds, or thousands of nodes), working at different layers of the protocol stack (e.g. L3 routers, L2 switches or L4 firewalls); moreover, rules of the forwarding tables may be complicated filters (for instance, performing a partial match on a specific part of a packet) and they can be mutually dependent (i.e. a rule is activated if and only if a previous rule does not match the incoming packet). In all the above cases, a deep inspection or a hand-made sanity check is practically unfeasible.

We provide some examples of forwarding problems generated because of forwarding tables' misconfiguration. When the forwarding rules are created thanks to a routing protocol, a manual modification may create new classes of packets that are rerouted until they eventually create a loop. When there are devices with *drop* rules, there can exist a set of "black-hole" nodes, where no packet at all is delivered to the original destination. Considering the example of packets that loop, detecting such a problem in a forwarding network is known to be an NP-complete problem [MKA+11], when general rules such as wildcard expressions are used as it stands for an SDN-network. Network administrators have always been interested in network diagnosis, and the resarch community showed significant activity about this topic: in the literature there are some tools that efficiently solve this problem in networks with thousands of forwarding rules (NetPlumber [KCZ+13], VeriFlow [KZZ+13], Anteater [MKA+11]).

Some of these tools make use of practical heuristics [KZZ+13, KCZ+13] to be computationally efficient, while others translate the problem detection into its equivalent SAT[1] problem (cf. Section 5.1 and Figure 5.1 for an example connecting loop detection and SAT problem) and solve the latter [MKA+11]. Both methods can be strongly affected by the typology of the network rules, by the number of nodes and the size of the forwarding tables.

We now begin to formalize the network verification problem. For the remainder of this part, a table of symbols is present at page 137.

5.1 Network Verification

For the remainder of this part of the thesis we focus on a key diagnosis task: detecting all possible forwarding loops. Our analysis and our main results can be extended to all types of verification problems. Given a network and nodes' forwarding tables, the problem consists in testing whether there exists a packet header h and a directed cycle in the network topology such that a packet with header h will indefinitely loop along the cycle.
The NP-completeness of this problem has been previously noted in [MKA+11]. Its hardness comes from the use of compact representations for predicate filters, that is the rules of the forwarding tables: the set of headers that match a rule is classically represented by a prefix in IP forwarding, a general wildcard expression in SDN, value ranges in firewall rules, or even a mix of such representations if several header fields are considered.

We first give a toy example of forwarding loop problem where the predicate filter of each rule is given by a wildcard expression, that is an ℓ-letter string in $\{1, 0, *\}^{\ell}$. Such an expression represents the set of all ℓ-bit headers obtained by replacing each $*$ of the expression by either 0 or 1. It is associated with the action to be taken on packets with header in that set (such packets or headers are said to match the rule): drop, forward to a neighbor or deliver locally. Figure 5.1 illustrates a one-node network with wildcard expressions of $\ell = 4$ letters. Rules are tested from top to bottom. All rules indicate to drop packets except the last one that forwards packets to the node itself. This network contains a forwarding loop if there exists a header $x_1x_2x_3x_4$ that matches no rule except the last one. For the sake of clarity, given the rule: $r = 110*$, saying that a header h does not match the rule r corresponds to having $h = x_1x_2x_3x_4$ with $x_1 = 0$, or $x_2 = 0$, or $x_3 = 1$. Since the last character in the rule expression is a "$*$", it does not affect the header matching. This one-node network thus has a forwarding loop iff the formula $(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3} \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2) \wedge (x_1)$ is satisfiable, which

[1] SAT is the abbreviation of *Boolean Satisfiability Problem*, shortened in SATISFIABILITY (hence SAT), and it is the problem of checking if a given boolean expression holds true.

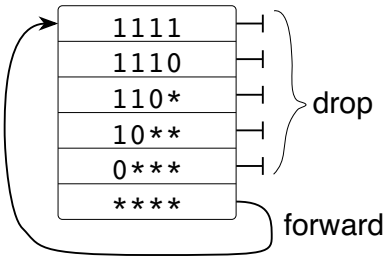


Figure 5.1: *Example of a one-node network without any forwarding loop.*

is not the case. This simple example can easily be generalized to reduce SAT to forwarding loop detection in networks with wildcard rules. It also points out a key problem: testing the emptiness of expressions such as $r_p \setminus \cup_{i=1..p-1} r_i$ where $r_1, \ldots, r_p$ are the sets associated to p rules. This expression identifies the set of all headers belonging to the rule r_p and not present in any of the previous rules r_i for i from 1 to $p-1$: if the expression is empty, then the rule r_p is never matched by any element, because all the possible headers could be "covered" by some of the previous rules.

One interesting observation is that the set of rules of a forwarding network defines a **σ-algebra**: given the set of all possible headers of l bits $H = \{0,1\}^l$, given a collection $\mathcal{R} = \{r_1, r_2, ..., r_n\}$ of subsets in H (the forwarding rules), which may include the empty set, the σ-algebra $\sigma(\mathcal{R})$ contains $\mathcal{R}$ and is closed under the set operations of *complement*, *union* and *intersection*. As packet headers in practical networks such as Internet typically have hundreds of bits, search of the header space is practically out of reach. The main challenge for solving such a problem thus resides in limiting the number of tests to perform. For that purpose, previous works [KZZ+13, KVM12] propose to consider sets of headers that match some predicate filters and do no match some others. Defining two headers as equivalent when they match exactly the same predicate filters, it then suffices to perform one test per equivalence class. These classes are indeed the **atoms** (the minimal non-empty sets) of the field of sets (the finite σ-algebra) generated by the sets associated to the rules.

Two challenges arise from this representation. A first challenge lies in efficiently identifying and representing these atoms. This would be fairly easy if both intersection and complement could be represented efficiently. In practice, most classical compact data-structures for sets of bit strings are closed under intersection but not under complement. For example, the intersection of two wildcard expressions, if not empty, can obviously be represented by a wildcard expression, but the complement of a wildcard expression is more problematic. Previous works overcome this difficulty by representing the complement of an ℓ-letter wildcard expression as the union of several wildcard expressions (up to ℓ). However, this can result in exponential blow-up and the tractability of these methods rely on various heuristics that do not offer rigorously proven guarantees.

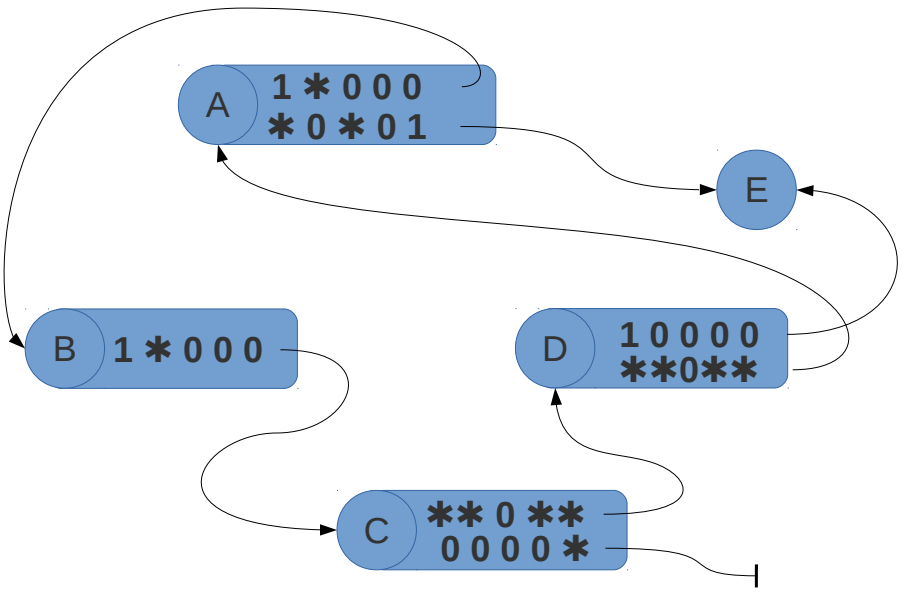
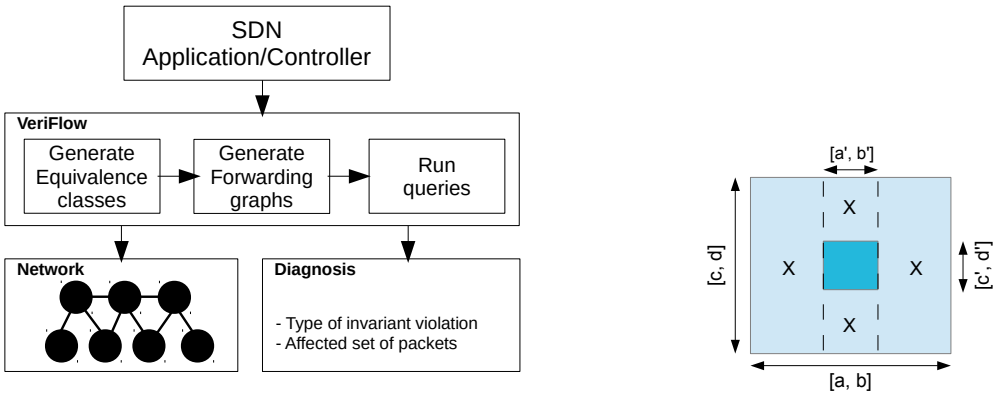


Figure 5.2: *Example of a network with forwarding tables. This network may have a loop depending on the packet header. The arrow indicates the direction of the forwarding action; arrows pointing to nothing stand for dropping a packet. All wildcard-based matching algorithms are considered. The header* 11000 *creates a loop.*

A second challenge lies in understanding the tractability of practical networks. One can easily design a collection of 2ℓ wildcard expressions that generates all the 2^ℓ possible singletons as atoms (all the ℓ-letters strings with only one non-$*$ letter). What does prevent such phenomenon in practice? The challenge that we want to address is to provide a property that intuitively fits with practical network and guarantees that the number of atoms does not blow up.

Finally, the representation of the forwarding rules, as well as the matching algorithm, may also affect the design of our framework. We identify four types of matching, as shown in Figure 5.2. *Exact matching* is when the bits of the header match the entire filter specified in the network rule; the first rule of the node D is an exact match. A *prefix matching* rule explicitly specifies only a prefix of the forwarding rule, while all remaining bits are marked with star; such a rule appears in the second rule of node C, where only the first 4 bits are specified and the last bit may be zero or one. IP routers are typically populated with prefix rules. The prefix matching generalizes in the *wildcard matching*, where any bit in any position may be marked with a star; rules of node A are all wildcards. SDN's current implementation might support wildcard rules. Finally, though not shown in the figure's example, another representation is the *(multi) range*. A *d-multirange* rule consists of several range specifications for different parts of the packets (commonly called *fields*). Firewalls typically use such matching rule (e.g. dropping all IP packets with destination ports in the range $[0, 80]$).



(a) *Architecture of VeriFlow (image from [KZZ+13])* **(b)** *VeriFlow class representation.*

Figure 5.3: *Architecture of VeriFlow and its representation of the header classes for* $R1 = \{[a, b]; [c, d]\}$ *and* $R2 = \{[a', b']; [c', d']\}$.

5.2 State of the art

Our inspiration for the development of our network verification model comes from the experience of both VeriFlow [KZZ+13] and NetPlumber/HSA [KCZ+13, KVM12]. Both tools adopt the approach of reducing the space of all possible elements to be verified by partitioning the l-bit header space $H = 0, 1^l$, and checking some properties for the obtained subset. They comprise a theoretical framework for the representation of header classes (or header expressions, for HSA), and a core library with some external adapters, which are used to manage the topologies and the forwarding tables of a SDN network. Adopting a different approach, other tools such as Anteater [MKA+11] use instead external SAT solvers to verify a desired network property, and therefore translate the verification problem in a boolean formula check.

In the following, we review both VeriFlow and NetPlumber/HSA, and give some details about Anteater.

VeriFlow VeriFlow [KZZ+13] is a tool designed for real-time verification of network-wide invariants such as the presence of loops or the reachability between two nodes. It is designed to work within an SDN network controller in order to obtain a screenshot of the network's forwarding tables and can check the validity of some invariant property at run-time. It efficiently performs the network validation task within the range of minutes thanks to its representation of the *header classes* for the given topology and the routers' forwarding rules.

VeriFlow runs on top of an SDN controller, which can show information about the underlying

topology and the forwarding rules of each node. A naive loop detection algorithm could consist in exploring the l-bit header space $H = \{0, 1\}^l$, performing a match on all nodes' rules. VeriFlow aims to reduce the number of tests to perform on predicate filters by identifying only the *sub-classes* affected by every new rule. Therefore, authors confine their verification activities to only those parts of the network whose actions may be influenced by a new update. This reduction translates in identifying the *equivalence classes* (ECs), defined as the set of packets that experience the same forwarding actions throughout the network.

An overview of the VeriFlow architecture is shown in Figure 5.3a. The first part of VeriFlow's verification algorithm is computing the equivalence classes for each new rule (that is, the set of headers affected by the new rule). An EC is detected by means of range comparison, and therefore is uniquely identified by its upper and lower bound[2]. The ECs are organized in a multi-dimensional trie, doubly linked with the corresponding rules and the nodes containing the specified rule. After the ECs are generated, a network forwarding graph is created for each EC in order to represent the network's forwarding behavior. Finally, the forwarding graph is traversed and some queries may be run in order to detect whether there is some kind of invariant violations.

Figure 5.3b shows an example of the VeriFlow's ECs representation. We consider two bi-dimensional range rules $R1 = \{[a, b]; [c, d]\}$ and $R2 = \{[a', b']; [c', d']\}$, where $R2 \subset R1$. When $R2$ is fully included in $R1$, it could be possible to consider a different behavior for only two classes of packets, each of them due to one specific rule. However, VeriFlow's representation generates several additional elements to represent the same header class. In fact, as shown in the Figure, VeriFlow is forced to compute the range differences, thus resulting in four additional elements to be verified. This affects the number of generated ECs, and so there are some unnecessary verification tests that could be avoided. We show more details about VeriFlow's EC computation in Section 6.4.3 at page 133.

From the experience of VeriFlow we keep in mind two key points: first, when d-level rule ranges are used, and additional sub-classes are unnecessarily represented, a d-exponential factor of tests may be required; second, the number of levels d can be tightly coupled with current Internet architecture (e.g. VeriFlow's source code comes with a hard-coded number of fields $d = 14$, which can represent the most important fields in standard SDN networks), thus limiting the flexibility of such a tool.

[2]VeriFlow reprents SDN rules which are in general multi-level wildcards. We show for simplicity only mono-dimensional or bi-dimensional rules. For instance, when $R1 = [0, 3[$ and $R2 = [2, 4[$, VeriFlow generates the ECs: $\{c_1 = [0, 2], c_2 = [2, 3], c_3 = [4]\}$.

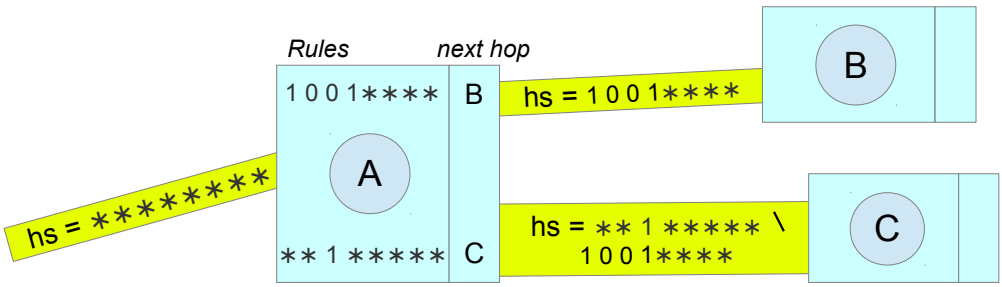


Figure 5.4: *NetPlumber/HSA wildcard expression generation. The topology contains 3 nodes $\{A, B, C\}$ and two rules; a starting wildcard expression is created and then propagated in the network when a match is found.*

NetPlumber/HSA NetPlumber [KCZ+13] is a tool for network verification which uses a geometric model of both the header space and the packet processing. It is based on the HSA framework [KVM12] and represents packets' headers as points in the l-bit geometric space $\{0, 1\}^l$. NetPlumber is connected to the SDN controller of the network that is to be verified, and translates the topology and the forwarding rules in a specific HSA syntax.

Tough having the same goals of VeriFlow, that is reducing the number of tests to perform in order to run queries on the network, NetPlumber does not calculate a set of classes to perform a query, but rather creates a wildcard rule (containing only star characters) that is then propagated and transformed on the fly, according to the matched forwarding rules. These transformations are called *transfer function*. Transfer functions are then applied for all network nodes and resulting expressions are propagated in the network graph.

This mechanism is described in Figure 5.4. The picture shows a network topology with 3 nodes and two rules in the forwarding table of node A. NetPlumber creates at first a wildcard expression representing all the header space (all bits are wildcard) and apply a transfer function on it using the first rule of A as filter. This is equivalent to performing a bit-wise AND operation between the two expressions. Finally, the result of the first expression is propagated to the next-hop node. Figure 5.4 shows that NetPlumber/HSA model includes as well set differences: the transfer function due the second rule of node A takes into account that the new generated expression should not contain the set of packets already verified by the first rule.

Since header classes are not defined in the HSA framework, we cannot measure the number of classes generated, and therefore we have to define some other parameter related to the computation time. This can be represented by the number of wildcard expressions generated by HSA. We remark that some of the generated expressions may be empty (i.e. the wildcard string does not represent any header set) and still be propagated by HSA: this is due to the lazy detection of emptiness for each propagation[3]. Additional details about NetPlumber expressions

[3]It is possible to force the emptiness test to be verified at every step, thus resulting in a reduced amount of

computation are shown in Section 6.4.2 at page 132.

NetPlumber can model any kind of wildcard matching, with a potentially unbounded number of bits per rule, resulting in a more flexible design with respect to VeriFlow. Moreover, authors assert that practical network can observe a small number of generated expressions thanks to a property called *linear fragmentation* (cf. Section 6.4 and Section 6.4.4 for more details). We get the inspiration from NetPlumber for two research directions: first, we develop our framework in order to be as much flexible as possible (i.e. model any kind of matching rules without being coupled with existing network architectures); second, we look for some property of practical networks which can allow us to prove that the number of tests could be bounded in real network environments.

Anteater Anteater [MKA+11] is a tool for network verification which exploits external SAT solver to run queries on a network topology. It tackles the classical invariant violations such as loop-detection and nodes reachability. Its design is tightly coupled with IPv4, and therefore it can analyze only IP forwarding rules containing string prefixes.

Anteater express the verification queries as SAT problems, that can be verified by external tools. It models the forwarding network as a tuple $G = (V, E, \mathbf{P})$, in which V is the set of vertexes, E is the set of edges and $\mathbf{P}$ is the representation of forwarding actions. In particular, $\mathbf{P}$ is a function with both domain E and codomain in $\{True, False\}$: for each edge (u, v), $\mathbf{P}(u, v)$ is the policy for packets traveling from u to v, represented as a boolean formula of predicate filters.

Anteater then verifies if the boolean formula corresponding to a specific policy check holds true. This can be done by means of external SAT-solving tools, or using a custom IP-specific algorithm for boolean formula's verification. We decided to avoid this SAT-translation approach and focus instead on reducing the number of tests by finding a good representation of the header classes.

5.3 Contributions

The main contributions of this part of the thesis are summarized in this section.

We propose a framework for a canonical representation of the atoms (i.e. the minimal non-empty sets) of the field of sets generated by a collection of sets. We provide an efficient algorithm for

expressions, at the cost of additional computation steps. However, we do not consider this for this experiment

computing such a representation. This tool is particularly suited when the intersection of two sets can be efficiently computed and represented. This is the case for forwarding networks, where the predicate filter associated to a rule can be seen as a compact data-structure representing the set of headers that match the rule. The header classes of the network (sets of headers matching the same rules) embrace all possible forwarding behaviors and their number measures how many tests are classically performed for forwarding loop detection. These classes are indeed the atoms of the field of sets generated by the collection of all predicate filters of the network. Following this equivalence, we first provide an efficient representation of atoms that allows to obtain the first polynomial time algorithm for loop detection in terms of number of classes. This contrasts with previous methods that can be exponential, even in simple cases with linear number of classes. We then introduce a notion of network dimension captured by the overlapping degree of forwarding rules. The values of this measure appear to be very low in practice and constant overlapping degree ensures polynomial number of header classes. Forwarding loop detection is thus polynomial in forwarding networks with constant overlapping degree. Our framework is described and analyzed in Chapter 6. A preliminary work has been published in [BDLPL$^+$15], and a following paper is currently under submission. We developed a software tool implementing our algorithms for loop detection, called *IHC*. We target to evaluate IHC's performance as future work (cf. page 143).

Chapter 6

Forwarding rule verification through atom computation

In this chapter we present the theoretical formalization of the problem of detecting loops in a network by analyzing the routers' forwarding tables. The main contribution of this chapter is twofold. First, we make a key algorithmic step by providing an efficient algorithm for computing an exact representation of the atoms of the field of sets generated by a collection of sets. The representation obtained is linear in the number of atoms and allows to test efficiently if an atom is included in a given set of the collection. The main idea is to represent an atom by the intersection of the sets that contain it. We avoid complement computations by using cardinality computations for testing emptiness. Our algorithm is generic and supports any data-structure for representing sets of ℓ-bit strings that allow intersection and cardinality computation in bounded time $O(T_\ell)$ for some value T_ℓ. It runs in polynomial time with respect to n and m, which are the number of sets and atoms respectively.

Rule repr.	Header cl.	Trivial	NetPlumber [KCZ+13]	VeriFlow [KZZ+13]	Our framework
T_ℓ-bounded	m	$O(T_\ell n n_G 2^\ell)$	–	–	$O(T_\ell n m^2 \log m + n n_G m)$
" o.d. $k_{\max}$	$m = O(n^{k_{\max}})$	"	–	–	$O(T_\ell n m + n_G n^{k_{\max}})$
ℓ-wildcard	$m \leq 2^{\min(\ell,n)}$	$O(\ell n n_G 2^\ell)$	$\Omega(\ell n_G 2^{\min(\ell,n)})$	$\Omega(n_G 2^{\min(\ell/2,n)})$	$O(\ell n m^2 + n n_G m)$
" o.d. $k_{\max}$	$m = O(n^{k_{\max}})$	"	$\Omega(\ell n_G 2^{\min(\ell,n)})$	$\Omega(n_G 2^{\min(\ell/2,n)})$	$O(\ell(n + k_{\max} 2^{k_{\max}})m + n_G n^{k_{\max}})$
d-multi-rng.	$m = O((2n)^d)$	$O(\ell n n_G (2n)^d)$	–	$\Omega((\frac{n}{d})^{d-1} n_G \frac{m}{d})$	$O(dnm^2 + n n_G m)$
" o.d. $k_{\max}$	$m = O(n^{k_{\max}})$	"	–	$\Omega((\frac{n}{d})^{d-1} n_G \frac{m}{d})$	$O(n^{k_{max}}(\ell k_{\max} 2^{k_{\max}} + \log^d n) + n_G n^{k_{\max}})$

Table 6.1: *Worst-case complexity of forwarding loop detection with n rules that generate m header classes in an n_G-node network, depending on rule set representation. T_ℓ-bounded: intersection and cardinality computations in $O(T_\ell)$ time; ℓ-wildcard: wildcard expressions with ℓ letters; d-multi-rng.: multi-ranges in dimension d. Additional hypothesis "o.d. $k_{\max}$" stands for overlapping degree of rules bounded by some constant $k_{\max}$. Our results are detailed in Section 6.3.2; NetPlumber and VeriFlow are analyzed in Section 6.4.2 and 6.4.3 respectively.*

The second contribution is related to real networks application: we provide a dimension parameter, the *overlapping degree* $k_{\max}$, which reflects the complexity of the collection of rules considered in a given forwarding network. It is defined as the maximum number of distinct rules (i.e. with pairwise distinct associated sets) that match a given header. This parameter constitutes a measure of complexity for the field of sets generated by a given collection of sets. In the context of practical hierarchical networks, we have the following intuitive reason to believe that this parameter is low: in such networks, more specific rules are used at lower levels of the hierarchy. We can thus expect that the overlapping degree is bounded by the number of layers of the hierarchy. Empirically, we observed a value within $5 - 15$ for datasets with hundreds to thousands of distinct multi-field rules, and $k_{\max} = 8$ for the collection of IPv4 prefixes advertised in BGP.

If the overlapping degree is constant, then the number of header classes is polynomially bounded: as a consequence, practical networks may be analyzed in a reasonable amount of time despite the NP-completeness of the verification problem. In addition, the algorithm we propose is tailored to take advantage of low overlapping degree $k_{\max}$, even without knowledge of $k_{\max}$. Table 6.1 provides a summary of the complexity results obtained for loop detection depending on how the sets associated to rules are represented.

We can summarize two main performance achievements of our algorithm for atom computation. First, it manages to remain polynomial in the number m of atoms even though the number of sets generated by intersection solely can be exponential in m with general rules. Second, the use of cardinality computations allows to avoid exponential blow-up (in contrast with previous work) but naturally induces a quadratic $O(m^2)$ term in the complexity. However, we manage to reduce it to $O(m)$ in the case of collections with constant overlapping degree.

The remainder of this chapter is organized as follows: in Section 6.1 we introduce the model; Section 6.2 describes how to represent the atoms of a field of sets; we then show our main results in Section 6.3, especially focusing on atom computation and its implications for forwarding loop detection that give the upper bounds listed in Table 6.1; in Section 6.4 we give more insight about the comparison of our results with previous works and justify the lower bounds presented in Table 6.1; finally, Section 6.5 concludes the chapter, providing some perspectives.

6.1 Model

We start this section by giving some definitions for our generic network model. We introduce as well the terminology that is used in this chapter.

6.1.1 Definitions

A *network instance* $\mathcal{N}$ is characterized by a graph $G = (V, E)$; each network node $u \in V$ is a *router* and has its own *forwarding table* $T(u)$. Every packet in the network has an associated *header* $h = \{b_1, b_2, ..., b_l\}, b_i \in \{0, 1\}$. The natural number ℓ represents the (fixed) bit-length of packet headers. Let H denote the set of all 2^ℓ possible headers (all ℓ-bit strings). We call the set H the *header space*. Given a space of elements H, we call *collection* a finite set of subsets of H. We also make use of the notion of *cardinality*: the cardinality of a set ($|s|$) is the number of elements in the set s. In the context of the header space, this translates to the number of headers contained in some set $s \subseteq H$. Cardinality computation is used to test if a set is empty or not, thanks to the equivalence: $|s| = 0 \iff s = \emptyset$.

Each forwarding table $T(u)$ is an ordered list of forwarding rules $(r_1, a_1), \ldots, (r_p, a_p)$. The number p is the size of the forwarding table, and it may be different for different routers. A generic *forwarding rule* (r, a), is made of a predicate filter r and an action a to apply on all packets whose header match the predicate. We say that a header h *matches* rule (r, a) when it matches predicate r (we may equivalently say that (r, a) matches h). We write $h \in r$ to emphasize the fact that r can be viewed as a compact data-structure encoding the set of headers that match it. This set is called the *rule set* associated to (r, a). For the ease of notation, we thus let r denote both the predicated filter of rule (r, a) and the associated set. Each a_i in the rule entry represents the action to be made on the packet whose header matches that rule. We consider three possible actions for a packet: forward to a neighbor, drop, or deliver (when the packet is arrived at destination). The priority of rules is given by the ordering of the rules: when a packet with header h arrives at node u, the first rule matched by h is applied. Equivalently, the rule (r_i, a_i) is applied when $h \in r_i \cap \overline{r_1} \cap \cdots \cap \overline{r_{i-1}}$, where $\overline{r}$ denotes the complement of r. When no match is found (i.e. $h \in \overline{r_1} \cap \cdots \cap \overline{r_p}$), the packet is dropped by default.

Given a header h, the *forwarding graph* $G_h = (V, E_h)$ of h represents the forwarding actions taken on a packet with header h. The forwarding graph is built in the following way: given some rule entry $(r_i, a_i) \in T(u)$, if we have $h \in r_i$ and $a_i = FORWARD(v)$, with $uv \in E_h$ and r_i is the first rule that matches h in $T(u)$, the corresponding action indicates to forward to v; therefore G_h consists of the set of all visited nodes $u \in V$ and all links $uv \in E_h$ traversed until either the packet is delivered, or it is dropped. The *forwarding loop detection* problem consists in deciding whether there exists a header $h \in H$ such that G_h has a directed cycle.

We make the simplifying assumption that the input port of an incoming packet is not taken into account in the forwarding decision of a node. In a more general setting, a node has a forwarding table for each incoming link. The model is not affected by this setting, except that we would have to consider the line-graph of G instead of G.

6.1.2 Header Classes

As introduced in Section 5.1, we can infer a natural relation of equivalence among headers with respect to rules: we assert that two headers are equivalent if they match exactly the same rules, that is if they belong to the same rule sets. In practice, this translates to the simple observation that two equivalent headers cause the corresponding packets to have exactly the same behavior in the network. The resulting equivalence classes partitions the header set H into nonempty disjoint subsets called *header classes*. Thanks to this equivalence relation, we can check any property of the network on a class-by-class basis instead of a header-by-header basis. Performing a verification problem in the space of the classes is guaranteed to prove a specific property for any h of the header space, but results in a smaller number of computation steps. A natural parameter for such a problem is the number m of header classes; it is thus interesting to quantify the difficulty of forwarding loop detection (or other similar network analysis problems) with respect to this parameter.

The header classes can be defined according to the collection $\mathcal{R}$ of rule sets of $\mathcal{N}$ (i.e. $\mathcal{R} = \{r \mid \exists u, a \text{ s.t. } (r,a) \in T(u)\}$). If $\mathcal{R}(h) \subseteq \mathcal{R}$ denotes the set of all rule sets associated to the rules matched by a given header h, then its header class is equal to $\left(\cap_{r\in\mathcal{R}(h)} r\right) \cap \left(\cap_{r\in\mathcal{R}\setminus\mathcal{R}(h)} \overline{r}\right)$ (with the convention $\cap_{r\in\emptyset} r = H$). Such sets are the atoms of the field of sets generated by $\mathcal{R}$. The computation of these sets is of relevance for this thesis, because performance depends also on the efficiency of this calculation.

6.1.3 Set representation

As we focus on the collection $\mathcal{R}$ of rule sets, we now detail our hypothesis on their representation. We assume that a data-structure $\mathcal{D}$ allows to represent some of the subsets of a space H. For the ease of notation, $\mathcal{D} \subseteq \mathcal{P}(H)$ also denotes the collection of subsets that can be represented with $\mathcal{D}$. We assume that $\mathcal{D}$ is closed under intersection: if s and s' are in $\mathcal{D}$, so is $s \cap s'$. We say that such a data-structure $\mathcal{D}$ for subsets of H is T_H*-bounded* when intersection and cardinality can be computed in time T_H at most: given the representation of $s, s' \in \mathcal{D}$, the representation of $s \cap s' \in \mathcal{D}$ and the size $|s|$ of s (as a binary big integer) can be computed within time T_H. As big integers computed within time T_H have $O(T_H)$ bits, this implies $|H| = 2^{O(T_H)}$: the bound T_H obviously depends on H. Intersection, inclusion test ($s \subseteq s'$), cardinality computation ($|s|$) and cardinal operations (addition, subtraction and comparison) are called *elementary set operations*. Under the T_H-bounded hypothesis, all these operations can be performed in $O(T_H)$ time ($s \subseteq s'$ is equivalent to $|s \cap s'| = |s|$).

In a forwarding network, we consider the header space $H = \{0,1\}^\ell$ of all ℓ-bit strings, which

may be plain or decomposed in several fields. If plain, a rule set is then typically represented by a wildcard expression or a range of integers. In both cases, they can be represented within 2ℓ bits and both representation are $O(\ell)$-bounded. We call *ℓ-wildcard* a string $e_1 \cdots e_\ell \in \{0, 1, *\}^\ell$. It represents the set $\{x_1 \cdots x_\ell \in \{0,1\}^\ell \mid \forall i, x_i = e_i \text{ or } e_i = *\}$. If H is decomposed into fields, any combinations of wildcard expressions and ranges can be used (either one for each field) and represented within $O(\ell)$ bits. However cardinality computations can take $\Theta(\ell \log \ell)$ time as multiplications of big integers are required. Such representation is thus $O(\ell \log \ell)$-bounded. Given d field lengths $\ell_1, \ldots, \ell_d$ with sum ℓ, we call *(d, ℓ)-multi-range* a cartesian product $[a_1, b_1] \times \cdots \times [a_d, b_d]$ of d integer ranges with $0 \leq a_i \leq b_i < 2^{\ell_i}$ for i in $1..d$. It represents the set $\{bin(x_1, \ell_1) \cdots bin(x_d, \ell_d) \mid (x_1, \ldots, x_d) \in [a_1, b_1] \times \cdots \times [a_d, b_d]\}$ where $bin(x_i, \ell_i)$ is the binary representation of x_i within ℓ_i bits.

6.1.4 Representation of a collection of sets

When manipulating a collection of p sets in $\mathcal{D}$, we assume that their representations are stored in a balanced binary search tree, allowing to dynamically add, remove or test membership of a set in $O(T_H \log p)$ time. Such operations are called *p-collection operations*. More efficient data-structures can be used for wildcard expressions and multi-ranges: we can use a balanced binary search tree when comparisons according to a total order can be performed. Such comparison can usually be obtained by comparing directly the binary representations themselves of the set in linear time (and thus $O(T_H)$ for sets with T_H-bounded representation). It is considered as an elementary set operation. In the case of wildcard expressions, the complexity of these operations can be reduced to $O(\ell)$ time by using a trie or a Patricia tree [Knu98]. Our algorithms will also make use of an operation similar to stabbing query[1] that we call *p-intersection* query. It consists in producing the list L_s of sets in a collection $\mathcal{R}$ of p sets that intersect a given query set s ($L_s = \{r \in \mathcal{R} \mid s \cap r \neq \emptyset\}$). We additionally require that the list L_s is topologically sorted w.r.t. inclusion. We say that p-intersection queries can be answered with *overhead* $(T_{inter}(p), T_{updt}(p))$ when dynamically adding or removing a set from the collection takes time $T_{updt}(p)$ at most and the p-intersection query for any set $s \in \mathcal{D}$ takes time $T_{inter}(p) + |L_s| T_H$ at most. In the case of d-dimensional multi-ranges, a segment tree allows to answer p-intersection queries with overhead $(O(\log^d p), O(\log^d p))$ [BCKO08, EM81]. In the case of wildcard expressions, a trie or a Patricia tree allows to answer p-intersection queries with overhead $(O(\ell p), O(\ell))$ (the whole tree has to be traversed in the worse case, but no sorting is necessary as the result is naturally obtained according to lexicographic order).

[1] In the field of computational geometry, the **stabbing problem** is the problem of detecting all n intervals (or segments) that intersect a given segment s (that may also be a simple point). It can be extended for any d-dimensional space [EMP+82].

6.2 Atoms generated by a collection of sets

Consider the collection $\mathcal{R}$ of subsets of the header space H. The *field of sets* $\sigma(\mathcal{R})$ generated by $\mathcal{R}$ is the (finite) σ-algebra generated by $\mathcal{R}$, which is the smallest collection closed under intersection, union and complement that contains $\mathcal{R} \cup \{\emptyset, H\}$. The *atoms* of $\sigma(\mathcal{R})$ are classically defined as the non-empty elements that are minimal for inclusion. We call them the atoms generated by $\mathcal{R}$ for the sake of simplicity. Let $\mathcal{A}(\mathcal{R})$ denote their collection. Note that for $a \in \mathcal{A}(\mathcal{R})$ and $r \in \mathcal{R}$, we have either $a \subseteq r$ or $a \subseteq \overline{r}$: in fact, otherwise $a \cap r$ and $a \cap \overline{r}$ would be non-empty elements of $\sigma(\mathcal{R})$ strictly included in a; this would translate in $\mathcal{A}(\mathcal{R})$ being non-minimal, going against the definition of atom. We can derive the following characterization of atoms (matching our definition of header classes when $\mathcal{R}$ is the collection of rule sets of a network):

$$\mathcal{A}(\mathcal{R}) = \left\{ a \neq \emptyset \mid \exists R \subseteq \mathcal{R}, a = \left(\cap_{r \in R} r\right) \cap \left(\cap_{r \in \mathcal{R} \setminus R} \overline{r}\right) \right\} \tag{6.1}$$

We now show that each atom $a = \left(\cap_{r \in R} r\right) \cap \left(\cap_{r \in \mathcal{R} \setminus R} \overline{r}\right)$ can be canonically represented by $c = \cap_{r \in R} r$ and then propose an incremental algorithm for computing such a representation for all atoms generated by a collection of sets.

6.2.1 Representing atoms by uncovered combinations

We first clarify the notion of representation: we say that a set $a' \subseteq H$ *inclusion-wise represents* an atom a generated by the collection $\mathcal{R}$ when $a \subseteq r \iff a' \subseteq r$ for all $r \in \mathcal{R}$. Equivalently, a' represents a when it has the same *containers*, i.e. they are contained in the same sets of $\mathcal{R}$: $\mathcal{R}(a') = \mathcal{R}(a)$ where $\mathcal{R}(s) = \{r \in \mathcal{R} \mid s \subseteq r\}$ denotes the sets in $\mathcal{R}$ that contain a set $s \subseteq H$. Note that such a representative set a' allows to efficiently compute $\mathcal{R}(a)$ when inclusion of a' can be tested efficiently.

As we consider that intersection of sets can be computed efficiently, we naturally consider the collection $\mathcal{C}(\mathcal{R}) \subseteq \sigma(\mathcal{R})$ of *combinations* defined as sets that can be derived by intersection from sets in $\mathcal{R}$:

$$\mathcal{C}(\mathcal{R}) = \{c \neq \emptyset \mid \exists R \subseteq \mathcal{R}, c = \cap_{r \in R} r\} \tag{6.2}$$

Our main idea is to determine whether a combination $c \in \mathcal{C}(\mathcal{R})$ inclusion-wise represents any atom. By definition of $\mathcal{R}(c)$, c is included in all sets in $\mathcal{R}(c)$ and none of the sets in $\mathcal{R} \setminus \mathcal{R}(c)$. The only atom it could inclusion-wise represent is thus $\left(\cap_{r \in \mathcal{R}(c)} r\right) \cap \left(\cap_{r \in \mathcal{R} \setminus \mathcal{R}(c)} \overline{r}\right)$ if ever this

set is not empty. We will solve such emptiness tests by means of cardinality computations in the sequel. But we can already state formally how combinations can represent atoms. For that purpose, a combination c is said to be *covered* in $\mathcal{R}$ when $c \subseteq \cup_{r\in\mathcal{R}\setminus\mathcal{R}(c)} r$. Informally, this happens when its non-containers (the sets in $\mathcal{R} \setminus \mathcal{R}(c)$) collectively contain it. Conversely, c is *uncovered* when it is not covered, or equivalently when $c \cap \left(\cap_{r\in\mathcal{R}\setminus\mathcal{R}(c)} \overline{r}\right) \neq \emptyset$.

The following proposition states that atoms can be represented by uncovered combinations.

Proposition 6.1 *Given a collection $\mathcal{R}$, each combination $c \in \mathcal{C}(\mathcal{R})$ can be associated to the set $a(c) = c \cap \left(\cap_{r\in\mathcal{R}\setminus\mathcal{R}(c)} \overline{r}\right)$. The collection $\mathcal{UC}(\mathcal{R}) = \{c \in \mathcal{C}(\mathcal{R}) \mid a(c) \neq \emptyset\}$ of uncovered combinations is in one to one correspondence with the collection $\mathcal{A}(\mathcal{R})$ of atoms generated by $\mathcal{R}$. Every combination $c \in \mathcal{UC}(\mathcal{R})$ inclusion-wise represents the atom $a(c)$. Moreover, $\mathcal{UC}(\mathcal{R})$ is the canonical representation of $\mathcal{A}(\mathcal{R})$ in the sense that it is the unique collection of combinations that inclusion-wise represent all atoms generated by $\mathcal{R}$.*

Proof The proof of Proposition 6.1 follows from the above discussion. First, we verify that if c is an uncovered combination, $a(c)$ is an atom: it suffices to observe that $c = \cap_{r\in\mathcal{R}(c)} r$ and to match $a(c) = c \cap \left(\cap_{r\in\mathcal{R}\setminus\mathcal{R}(c)} \overline{r}\right)$ with the atom characterization given by Equation (6.1) using $R = \mathcal{R}(c)$. Similarly, if $a = (\cap_{r\in R} r) \cap \left(\cap_{r\in\mathcal{R}\setminus R} \overline{r}\right)$ is an atom for some $R \subseteq \mathcal{R}$, the combination $c(a)$ satisfies $\mathcal{R}(c(a)) = R$ which implies $a(c(a)) = a$. In particular, as $a \neq \emptyset$, $c(a)$ is uncovered. □

Example Consider the 8-element space $H = \{0..7\}$ and the collection $\mathcal{R} = \{r_1 = \{0..4\}, r_2 = \{1..5\}, r_3 = \{2..6\}, r_4 = \{3\}\}$. The atoms generated by $\mathcal{R}$, as defined in Equation 6.1, are $\mathcal{A}(\mathcal{R}) = \{\{0\}, \{1\}, \{2,4\}, \{3\}, \{5\}, \{6\}, \{7\}\}$, where:

$\{0\} = r_1 \cap \bar{r_2} \cap \bar{r_3} \cap \bar{r_4}$, $\{1\} = r_1 \cap r_2 \cap \bar{r_3} \cap \bar{r_4}$, $\{2,4\} = r_1 \cap r_2 \cap r_3 \cap \bar{r_4}$, $\{3\} = r_1 \cap r_2 \cap r_3 \cap r_4$, $\{5\} = \bar{r_1} \cap r_2 \cap r_3 \cap \bar{r_4}$, $\{6\} = \bar{r_1} \cap \bar{r_2} \cap r_3 \cap \bar{r_4}$, and $\{7\} = \bar{r_1} \cap \bar{r_2} \cap \bar{r_3} \cap \bar{r_4}$.

Figure 6.1 shows a visual representation of this configuration. Due to the *complement* operations, the atoms can be harder to represent than the rules they are generated from: in fact, in Figure 6.1, all rules are ranges, but the atom $\{2,4\}$ is not. In Figure 6.1, there are 8 distinct combinations, following from Equation 6.2 and Proposition 6.1: r_1, $r_1 \cap r_2$, $r_1 \cap r_3$, r_4, $r_2 \cap r_3$, r_3, H and r_2.

The property of the characterization of Proposition 6.1 we are interested in is that it allows to efficiently test whether a set $r \in \mathcal{R}$ contains an atom $a \in \mathcal{A}(\mathcal{R})$: given a combination c that represents a, $a \subseteq r$ is equivalent to $c \subseteq r$. This comes from the fact that every uncovered combination c has same containers as $a(c)$. (If it was not the case, $a(c) = \emptyset$ and c is covered.)

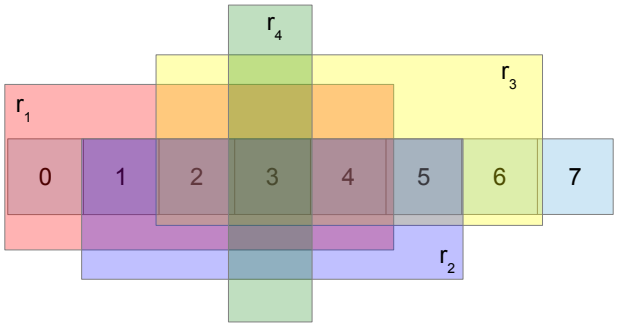


Figure 6.1: *Example of the representation of the 8-element space $H = \{0..7\}$, and the aforementioned rules.*

This explains the importance of determining $\mathcal{UC}(\mathcal{R})$, and in particular to separate covered combinations from uncovered ones.

Informally, we can associate each atom $a \in \mathcal{A}(\mathcal{R})$ with the combination $c(a) := \cap_{r \in \mathcal{R}(a)} r$, and this corresponds to the "positive" part of the characterization from Equation 6.1: the complement set operation is therefore not needed for our model.

6.2.2 Overlapping degree of a collection

Our representation is naturally associated to the following measure of complexity of a collection $\mathcal{R}$. We define the *overlapping degree* $k_{\max}$ of $\mathcal{R}$ as the maximum number of containers of an element, that is $k_{\max} = \max_{h \in H} |\mathcal{R}(\{h\})|$. Note that all elements within an atom have same containers in $\mathcal{R}$ and that a set cannot have more containers than any of its elements. We thus have:

$$k_{\max} = \max_{s \subseteq H} |\mathcal{R}(s)| = \max_{a \in \mathcal{A}(\mathcal{R})} |\mathcal{R}(a)|$$

As any atom a can be expressed as $a = (\cap_{r \in R} r) \cap \left(\cap_{r \in \mathcal{R} \setminus R} \overline{r}\right)$ where $R = \mathcal{R}(a)$ is the set of containers of a, the number of atoms is obviously bounded by $\binom{n}{k_{\max}}$ where $n = |\mathcal{R}|$ denotes the number of sets in $\mathcal{R}$. The overlapping degree of a collection thus measures its complexity in terms of number of atoms it may generate.

We similarly define the *average overlapping degree* $\overline{k}$ as the average number of containers of an atom:

$$\overline{k} = \frac{\sum_{a \in \mathcal{A}(\mathcal{R})} |\mathcal{R}(a)|}{|\mathcal{A}(\mathcal{R})|}$$

.

We obviously have $\overline{k} \leq k_{\max}$. Given a collection $\mathcal{R}$, we will also consider the average overlapping degree $\overline{K}$ of combinations, that is the average overlapping degree of $\mathcal{C}(\mathcal{R})$. Note that $\mathcal{C}(\mathcal{R})$ has

the same collection of atoms as $\mathcal{R}$ and that a combination $c = \cap_{r \in \mathcal{R}(c)} r$ containing an atom a must satisfy $\mathcal{R}(c) \subseteq \mathcal{R}(a)$. The overlapping degree of $\mathcal{C}(\mathcal{R})$ is thus at most 2^k and we always have $\overline{K} \leq 2^k$. In the example from Figure 6.1, it is easy to verify that we have $k = 4$, $\overline{k} = 13/7$ and $\overline{K} = 4$.

In real datasets we observe that both k and $\overline{K}$ are in the range $[2, 15]$ while $\overline{k}$ is in the range $[1.5, 5]$ (these include Inria firewall rules, Stanford forwarding tables provided by Kazemian et al. [Kaz] and IPv4 prefixes announced at BGP level collected from the RouteViews project [Rou]).

6.3 Incremental computation of atoms

We can now state our main result concerning the computation of the atoms generated by a collection of sets. Section 6.3.1 explain the incremental computation of atoms of a collection of sets. Section 6.3.2 describes how the our incremental algorithm can be used to detect forwarding loops.

6.3.1 Computation of atoms generated by a collection of sets

We first provide some notions about cardinality computation and the effect of the incremental add of rules in Section 6.3.1. We give a basic algorithm for the incremental atoms computation in Section 6.3.1.1, and refine this result in Section 6.3.1.2 where we take into account the overlapping degree of the network. Finally, we show and prove our main theorem in Section 6.3.1.3, giving the time boundaries of our representation.

The incremental update requires that when a new rule r is added to the set, it should be possible to compute the new atoms without recalculating from the beginning. The following lemma formally states that uncovered combinations of $\mathcal{R} \cup \{r\}$ can be obtained from $\mathcal{UC}(\mathcal{R})$. The use of an incremental algorithm allows to avoid exponential blow-up even in cases where the number of combinations can be exponential in the number m of atoms. This happens with the collection of complements of the singletons of an n-element set (cf. Section 6.4.2).

Lemma 6.1 *Given a new rule $r \subseteq H$, the collection $\mathcal{UC}(\mathcal{R}')$ of uncovered combinations of $\mathcal{R}' = \mathcal{R} \cup \{r\}$ can be obtained by intersecting uncovered combinations in $\mathcal{UC}(\mathcal{R})$ with r. More precisely, we have $\mathcal{UC}(\mathcal{R}') \subseteq \mathcal{UC}(\mathcal{R}) \cup \{c \cap r | c \in \mathcal{UC}(\mathcal{R})\}$.*

Proof Consider an uncovered combination $c' \in \mathcal{UC}(\mathcal{R}')$. Let c be the intersection of the containers of c' in $\mathcal{R}$: $c = \cap_{r \in \mathcal{R}(c')} r$. We have either $c' = c$ if $\mathcal{R}'(c') = \mathcal{R}(c')$ or $c' = c \cap r$ if

$\mathcal{R}'(c') = \mathcal{R}(c') \cup r$. To conclude, it is thus sufficient to show that c is uncovered in $\mathcal{R}$. This follows from $c' \subseteq c$ and $\mathcal{R}'(c') \subseteq \mathcal{R}(c) \cup \{r\}$: the non-containers of c in $\mathcal{R}$ are also non-containers of c' in $\mathcal{R}'$ and if c was covered, so would be c'.

□

We base our model on cardinality computation, that is used to test the emptiness of the atom set. We want to compute the cardinality of $a(c) = r_1 \cap r_2 \cap \ldots \cap \overline{r_k} \cap \overline{r_{k+1}} \cap \ldots \cap \overline{r_n}$ from its representation $c = r_1 \cap r_2 \cap \ldots \cap r_n$. The following Lemma expresses any combination as a disjoint union of atoms, and justify the incremental computation of atom cardinalities.

Lemma 6.2 *Given a combination collection $\mathcal{C}' \subseteq \mathcal{C}(\mathcal{R})$ containing $\mathcal{UC}(\mathcal{R})$, we have $d = \cup_{c \in \mathcal{C}' | c \subseteq d} a(c)$ for all $d \in \mathcal{C}'$. This union is disjoint and we have $|a(d)| = |d| - \sum_{c \in \mathcal{C}' | c \subsetneq d} |a(c)|$.*

Proof Reminding that any combination d includes $a(d) = d \cap \left(\cap_{r \in \mathcal{R} \setminus \mathcal{R}(d)} \overline{r}\right)$, we have $\cup_{c \in \mathcal{C}' | c \subseteq d} a(c) \subset d$. Conversely, consider $h \in d$. The sets of $\mathcal{R}$ containing h are $\mathcal{R}(\{h\})$, and we have $h \in a(c)$ for $c = \cap_{r \in \mathcal{R}(\{h\})} r$. As $\mathcal{R}(d) \subseteq \mathcal{R}(\{h\})$ (the sets that contain d also contain h) and $d = \cap_{r \in \mathcal{R}(d)} r$, we have $c \subseteq d$. Hence $d \subset \cup_{c \in \mathcal{C}' | c \subseteq d} a(c)$. This union is disjoint as each $a(c)$ is either an atom or is empty.

□

6.3.1.1 Basic algorithm for atom computation

We first propose a basic algorithm for updating the collection $\mathcal{UC}$ of uncovered combinations of a collection $\mathcal{R}$ when a set r is added to $\mathcal{R}$. The main idea is that after adding r to $\mathcal{R}$, the only new uncovered combinations that can be created are intersections of pre-existing uncovered combinations with r (see Lemma 6.1). We thus first add to $\mathcal{UC}$ the combinations $c \cap r$ for $c \in \mathcal{UC}$. As this may introduce covered combinations, we then compute the atom size $c.atsize = |a(c)|$ for each combination c. It is then sufficient to remove any combination c with $c.atsize = 0$ to finally obtain $\mathcal{UC}(\mathcal{R} \cup \{r\})$. This atom size computation is possible because we have $d = \cup_{c \in \mathcal{UC} | c \subseteq d} a(c)$ for all $d \in \mathcal{UC}$. (see Lemma 6.2). As this union is disjoint, we have $|a(d)| = |d| - \sum_{c \in \mathcal{UC} | c \subsetneq d} |a(c)|$. We thus compute the inclusion relation between combinations and store in $c.sup$ the combinations that strictly contain c. Initializing $c.atsize$ to $|c|$ for all c, we then scan all combinations c by non-decreasing cardinality (or any topological order for inclusion) and subtract $c.atsize$ from $d.atsize$ for each $d \in c.sup$. A simple proof by induction allows to prove that $c.atsize = |a(c)|$ when c is scanned. The whole process in summarized in Algorithm 6.1.

Algorithm 6.1: Add a set r to a collection $\mathcal{R}$ and update the collection $\mathcal{UC} = \mathcal{UC}(\mathcal{R})$ of its uncovered combinations accordingly.

Procedure *basicAdd*$(r, \mathcal{R}, \mathcal{UC})$
- $\mathcal{UC}' := \mathcal{UC} \cup \{c \cap r \mid c \in \mathcal{UC}\}$;
- **For each** $c \in \mathcal{UC}'$ **do**
  - $c.atsize := |c|$;
  - $c.sup := \{d \in \mathcal{UC}' \mid c \subsetneq d\}$;
- **For each** $c \in \mathcal{UC}'$ *in non-decreasing cardinality order* **do**
  - **For each** $d \in c.sup$ **do** $d.atsize := d.atsize - c.atsize$;
  - ;
- $\mathcal{UC} := \mathcal{UC}' \setminus \{c \in \mathcal{UC}' \mid c.atsize = 0\}$;

The correctness of Algorithm 6.1 follows from the two above remarks (that is Lemma 6.1 and Lemma 6.2). Its main complexity cost comes from intersecting r with each combination and computing the inclusion relation between combinations, that is $O(nm)$ and $O(m^2)$ elementary set operations respectively. Starting from $\mathcal{UC} = \{H\}$ and incrementally applying Algorithm 6.1 to each set in $\mathcal{R}$ thus allows to obtain $\mathcal{UC}(\mathcal{R})$ with $O(nm^2)$ elementary set operations.

We thus obtain the following theorem :

Theorem 6.1 *Given a space set H and a collection $\mathcal{R}$ of n subsets of H, the collection $\mathcal{UC}(\mathcal{R})$ of combinations that canonically represent the atoms $\mathcal{A}(\mathcal{R})$ can be incrementally computed with $O(\min(n + k\overline{K} \log m, nm)m)$ elementary set operations where: m is the number of atoms generated by $\mathcal{R}$; k is the overlapping degree of $\mathcal{R}$; $\overline{K}$ is the average overlapping degree of $\mathcal{C}(\mathcal{R})$; $\overline{k}$ is the average overlapping degree of $\mathcal{R}$. Within this computation, each combination $c \in \mathcal{UC}(\mathcal{R})$ can be associated to the list $\mathcal{R}(c)$ of sets in $\mathcal{R}$ that contain c. If sets are represented by ℓ-wildcard expressions (resp. (d, ℓ)-multi-ranges), the representation can be computed in $O(\ell \min(n + k\overline{K}, nm)m)$ (resp. $O(\ell \min(\overline{k} \log^d m + k\overline{K}, nm)m)$) time.*

In the following, we refine this result taking into account the overlapping degree. We define an optimized algorithm to efficiently compute atoms, and model the time boundaries in Theorem 6.2, which we enunciate and prove in Section 6.3.1.3 at page 128.

6.3.1.2 Optimized algorithm for atom computation

To derive better bounds for low overlapping degree k, we propose a more involved algorithm that maintains *c.sup* and *c.atsize* from one iteration to another and makes only the necessary updates. This requires to handle several subtleties to enable lower complexity.

We similarly start by computing the collection $Inter = \{c \in \mathcal{UC} \mid c \cap r \neq \emptyset\}$ of combinations intersecting r. A first subtlety comes from the fact that several combinations c may result in the same $c' = c \cap r$. However, we are only interested in the combination c which is minimal for inclusion that we call the *parent* of c'. The reason is that $c'.sup$ can then be computed from $c.sup$. The parent is unique unless c' is covered in which case c' is marked as covered and discarded (see the argument for $c'.sup$ computation later on). To obtain right parent information, we thus process all $c \in Inter$ by non-decreasing cardinality. The produced combinations $c' = c \cap r$ such c' were not in $\mathcal{UC}$ are called *new* combinations. Their atom size is initialized to $c'.atsize = |c'|$. See the "Parent computation" part of Algorithm 6.2.

We then remark that we only need to compute (or update) $c.sup$ for combinations that include r, which we store in a set *Incl*. We also note that $c.atsize$ needs to be computed when c is new and updated when c is the parent of a new combination. A second subtlety resides in computing (or updating) $c.sup$ only when c is not covered that is when $c.atsize$ (after computation) appears to be non-zero. As the computation of $c.sup$ lists is the most heavy part of the computation, this is necessary to enable our complexity analysis. For that purpose, we scan *Incl* by non-decreasing cardinality so that the correct value of $c.atsize$ is known when c is scanned similarly as in Algorithm 6.1. However, we avoid any useless computation when $c.atsize$ is zero. Otherwise, we compute (or update) $c.sup$ and decrease $d.atsize$ by $c.atsize$ from $d \in c.sup$ for adequate d: if c and d where both in $\mathcal{UC}$, this computation has already been made; it is only necessary when d is new or when d is the parent of c. We optionally maintain for each combination c a list $c.cont$ that contain the list of sets $r \in \mathcal{R}$ that contain c (Such lists are not necessary for the computation but they are useful for loop detection as detailed in Section 6.3.2). See the "Atom size computation" part of Algorithm 6.2.

A last critical point resides in the computation of $c'.sup$ for each new combination c'. The $c'.sup$ list can be obtained from $c'.parent.sup$ by copying and also intersecting elements of $c'.parent.sup$ with r. This is sufficient: for $d \in \mathcal{UC}$ such that $c' \subsetneq d \cap r$, we can consider $c = c'.parent \cap d$. If $c \in \mathcal{UC}$, $c'.parent = c$ by minimality of $c'.parent$ and we thus have $d \in c'.parent.sup$. The case where $c \notin \mathcal{UC}$ and $d \notin c'.parent.sup$ cannot happen as it would imply that two different combinations $c_1 = c'.parent$ and $c_2 \subseteq c$ generate c' by intersection with r ($c_1 \cap r = c_2 \cap r = c'$) and are both minimal for inclusion. In such case, $c_1 \cap c_2$ was covered in $\mathcal{R}$ and so would be c' in $\mathcal{R}$ (and also in $\mathcal{R} \cup \{r\}$). That is why such combination c' are already discarded during parent computation. On the other hand, the list $c.sup$ of a combination $c \in \mathcal{UC}$ can be updated by intersecting elements of $c.sup$ with r: when $c \subsetneq d \cap r$ for $c \subseteq r$, we have $c \subsetneq d$.

Finally, combinations c with $c.atsize = 0$ are discarded and removed from $b.sup$ list of remaining combinations as detailed in the "Remove covered combinations" part of Algorithm 6.2.

Algorithm 6.2: Add a set r to a collection $\mathcal{R}$ and update the collection $\mathcal{UC}$ of its uncovered combinations accordingly.

Procedure $add(r, \mathcal{R}, \mathcal{UC} = \mathcal{UC}(\mathcal{R}))$

$New := \emptyset$; $Incl := \emptyset$;;

`/* ---------- Parent computation ---------- */`

$Inter := \{c \in \mathcal{UC} \mid c \cap r \neq \emptyset\}$;

Sort *Inter* by non-decreasing cardinality.;

For each $c \in Inter$ **do**

  $c' := c \cap r$;

  If $c' \notin Incl$ **then**

    If $c' \notin \mathcal{UC}$ **then**

      $\mathcal{UC} := \mathcal{UC} \cup \{c'\}$; $New := New \cup \{c'\}$;;

      $c'.atsize := |c'|$; $c'.sup := \{\}$; $c'.cont := \{\}$ `/* Updated later. */`

    $c'.parent := c$; $c'.covered := false$; $Incl := Incl \cup \{c'\}$;;

  else

    If $c'.parent \not\subseteq c$ **then** $c'.covered := true$;

Remove from *Incl*, *New* and $\mathcal{UC}$ all c such that $c.covered = true$.;

`/* ---------- Atom size computation ---------- */`

Sort *Incl* by non-decreasing cardinality.;

For each $c \in Incl$ **do**

  If $c.atsize > 0$ **then**

    `/* Adjust` $c.sup$, $c.cont$ `and update` $d.atsize$ `for impacted` $d \supsetneq c$: `*/`

    If $c \in New$ **then**

      $c.parent.atsize := c.parent.atsize - c.atsize$;

      $c.sup := \{c.parent\} \cup c.parent.sup$;

      $c.cont := c.parent.cont$;

    $c.sup := c.sup \cup \{d \cap r \mid d \in c.sup$ and $d \cap r \in Incl \setminus \{c\}\}$;

    $c.cont := c.cont \cup \{r\}$;

    For each $d \in c.sup$ *s.t.* $d \in New$ **do**

      $d.atsize := d.atsize - c.atsize$;

`/* ---------- Remove covered combinations ---------- */`

For each $c \in Incl$ **do**

  Remove from $c.sup$ any d such that $d.atsize = 0$.;

  If $c.atsize = 0$ **then** $\mathcal{UC} := \mathcal{UC} \setminus \{c\}$; $Incl := Incl \setminus \{c\}$; ;

  ;

  If $c \in New$ *and* $c.parent.atsize = 0$ **then** $\mathcal{UC} := \mathcal{UC} \setminus \{c.parent\}$;

6.3.1.3 Time boundaries for the atoms computation

We now present our main result for this Section. For the sake of simplicity of asymptotic expressions, we make the very loose assumption that $\ell = o(m)$ and $n \leq m$. (We are mainly interested in the case where m is large. Note also that examples with $m < n$ would be very

peculiar.)

Proposition 6.2 *Algorithm 6.2 allow to dynamically update the collection $\mathcal{UC}$ of uncovered combinations of a collection $\mathcal{R}$ using $O(m + \sum_{a \in A_r} |\mathcal{UC}'(a)| \log m)$ elementary set operations when a rule r is added to $\mathcal{R}$, where m denotes the number of atoms of $\mathcal{R}$, $A_r = \{a \in \mathcal{A}(\mathcal{R}') \mid a \subseteq r\}$ denotes the atoms of $\mathcal{R}'$ included in r, and $\mathcal{UC}'(a)$ denotes the uncovered combinations of $\mathcal{R}'$ that contain a.*

More precisely, if the data-structures used for representing sets and collections of sets enable elementary set operations within time T_{set}, p-collection operations within time $T_{coll}(p)$ and p-intersection queries with overhead $(T_{inter}(p), T_{updt}(m))$, then the update of $\mathcal{UC}$ can be performed in $O(T_{inter}(m) + |A_r|\, T_{updt}(m) + (T_{set} + T_{coll}(m)) \sum_{a \in A_r} |\mathcal{UC}'(a)|)$ time.

Proof As discussed before the correctness of Algorithm 6.2 for obtaining uncovered combinations after adding set r to a collection $\mathcal{R}$ from $\mathcal{UC} = \mathcal{UC}(\mathcal{R})$ and $\{c \cap r \mid c \in \mathcal{UC}\}$ results from Lemma 6.1. For a new combination c, the $c.sup$ list is obtained from $c.parent.sup$ and $c.sup$ is updated similarly for $c \in \mathcal{UC}$ such that $c \subseteq r$. The correctness of this approach has already been discussed in Subsection 6.3. We develop here the key argument for ignoring a combination $c' = c \cap r$ when it is produced by several minimal elements $c_1, \ldots, c_i \in \mathcal{UC}$ such that $c_j \cap r = c'$ for j in $1..i$. If this happens, we know that $\cap_{j \in 1..i} c_j$ is not in $\mathcal{UC}$, meaning that it is covered in $\mathcal{R}$ and so is $c' \subseteq \cap_{j \in 1..i} c_j$ in $\mathcal{R} \cup \{r\}$. We can thus safely eliminate c' in the first phase of the algorithm. For the remaining new combinations c', the parent c of c' is the unique combination $c \in \mathcal{UC}$ such that $c \cap r = c'$ and which is minimal for inclusion.

The correctness of the atom cardinality computation follows by induction on the number combinations in *Incl* processed so far in the corresponding for loop. Consider a newly created combination c. The initial value of $c.atsize$ is $|c|$. Assuming that the correct value $b.atsize$ has been obtained for b processed before c and $c \in b.sup$, $|a(b)|$ has been subtracted from $c.atsize$ and Lemma 6.2 implies that $c.atsize = |a(c)|$ when we consider c in the for loop. For c already in $\mathcal{UC}$ before adding r and for $b \subseteq c$ processed before c, $b.atsize$ has been subtracted from $c.atsize$ only for newly created b. For $b \in \mathcal{UC}$, $|a(b)|$ may have decreased but this difference is compensated by $\sum_{b' \in New | b' \subsetneq b} |a(b')|$. This is the reason why Algorithm 6.2 updates only $c.parent.atsize$ besides $d \in c.sup$ such that $d \in New$. The correctness of the atom cardinality computation implies that all covered combinations are removed and the correctness of Algorithm 6.2 follows.

□

Complexity analysis We now analyze the complexity of Algorithm 6.2. The bound in terms of elementary set operations is obtained when balanced binary search tree (BST for short) are

used to store the various collections of sets (i.e. $\mathcal{UC}$, *Incl*, *New* and *c.sup* for $c \in \mathcal{UC}$). When adding a set r, finding the combinations in $\mathcal{UC}$ that intersect r is a m-intersection query and can be performed in $O(T_{inter}(m) + |Inter|T_{set})$ time or $O(m \log m)$ set operations using BST (sorting is only necessary in that case). The collection *Incl* is then constructed in $O(|Inter| \log m)$ operations with BST or $O(|Inter|(T_{set} + T_{coll}(m)))$ with appropriate data-structures. Removing combinations c such that $c.covered = true$ is just a matter of scanning *Incl* again and can be done within the same complexity. Let $\mathcal{I}$ denote the combinations included in *Incl* at that point (just before cardinality computations). The computation of *c.sup* for $c \in \mathcal{I}$ is done only when $c.atsize > 0$, i.e. only if c represents one of the atoms in A_r. we thus have $|I| \leq |A_r|$. This requires at most $O(|c.parent.sup|)$ operations. Note that for each uncovered combination $d \in c.parent.sup$ yields at least one uncovered combination in *c.sup* (d itself or $d \cap r$ or both). We thus have $|c.parent.sup| \leq |\mathcal{UC}'(a(c))|$. The overall computation of *sup* lists can thus be performed within $O(\sum_{a \in A_r} |\mathcal{UC}'(a)| \log m)$ set operations with BST and $O((T_{set} + T_{coll}(m)) \sum_{a \in A_r} |\mathcal{UC}'(a)|)$ time with appropriate data-structures. The computation of class cardinalities and the removal of covered combinations from the *sup* lists have same complexity. Removal of covered combinations from $\mathcal{UC}$ and *Incl* takes $O(|A_r|)$ collection operations and fits within the same complexity bound. Additional cost of $|A_r| T_{updt}(m)$ is necessary when maintaining data-structures enabling efficient m-intersection queries. The whole algorithm can thus be performed in $O(T_{inter}(m) + |A_r| T_{updt}(m) + |Inter|(T_{set} + T_{coll}(m)) + (T_{set} + T_{coll}(m)) \sum_{a \in A_r} |\mathcal{UC}'(a)|)$ time or using $O(m + |Inter| \log m + \sum_{a \in A_r} |\mathcal{UC}'(a)| \log m)$ elementary set operations with BST.

To achieve the proof of complexity of Algorithm 6.2, we show $|Inter| \leq \sum_{a \in A_r} |\mathcal{UC}'(a)|$. Consider a combination $c \in \mathcal{UC}$ that intersects r. Then c can be associated to an atom *c.atm* of A_r included in $c \cap r$ (such atoms exist according to Lemma 6.2). For any atom $a \in A_r$, let *a.par* denote the atom in $\mathcal{A}(\mathcal{R})$ that contains a (we have $a = a.par$ or $a = a.par \cap r$). Now for $c \in Inter$, consider the atom $a = c.atm \in A_r$. As $a.par \in \mathcal{A}(\mathcal{R})$ is an atom and $c \in \mathcal{C}(\mathcal{R})$ is a combination intersecting *a.par*, we have $a.par \subseteq c$ and $c \in \mathcal{UC}(a.par)$ is one of the combinations in $\mathcal{UC}(\mathcal{R})$ that contains *a.par*. For each such combination c, $a(c)$ intersects r or $\overline{r}$ (or both), and c or $c \cap r$ is uncovered in $\mathcal{R}' = \mathcal{R} \cup \{r\}$. Both contain a and we have $|\mathcal{UC}(a.par)| \leq |\mathcal{UC}'(a)|$. We can thus write $|Inter| = \sum_{a \in A_r} |\{c \in Inter \mid c.atm = a\}| \leq \sum_{a \in A_r} |\mathcal{UC}(a.par)| \leq \sum_{a \in A_r} |\mathcal{UC}'(a)|$. □

Proposition 6.2 represents the time boundaries of iterative application of Algorithm 6.2 (and Algorithm 6.1 as well). The following theorem models the incremental atom computation for a collection of rules under the assumption of a constant overlapping degree.

Theorem 6.2 *Given a space set H and a collection $\mathcal{R}$ of n subsets of H, the collection $\mathcal{UC}(\mathcal{R})$ of combinations that canonically represent the atoms generated by $\mathcal{R}$ can be incrementally com-*

puted with $O(\min(n + k\overline{K}\log m, \overline{k}m\log m, nm)m)$ *elementary set operations where* m *denotes the number of atoms generated by* $\mathcal{R}$*,* k *denotes the overlapping degree of* $\mathcal{R}$*,* $\overline{k}$ *denotes the average overlapping degree of* $\mathcal{R}$ *and* $\overline{K}$ *denotes the average overlapping degree of* $\mathcal{C}(\mathcal{R})$*.*

More precisely, if the data-structures used for representing sets and collections of sets enable elementary set operations within time T_{set}*, p-collection operations within time* $T_{coll}(p)$ *and p-intersection queries with overhead* $(T_{inter}(p), T_{updt}(m))$*, then the representation of the atoms generated by* $\mathcal{R}$ *can be computed in* $O(nT_{inter}(m)+\overline{k}mT_{updt}(m)+\min(k\overline{K}, km)m(T_{set}+T_{coll}(m)))$ *time.*

Proof From Proposition 6.2, the overall complexity of atom computation is:

$$O\left(\sum_{i=1..n}\left(m_i + \sum_{a\in A_i}|\mathcal{UC}_i(a)|\right)\log m\right)$$

set operations where m_i denotes the number of atoms in $\mathcal{A}(\{r_1,\ldots,r_{i-1}\})$ and A_i denotes the atoms of $\mathcal{A}(\{r_1,\ldots,r_i\})$ included in r_i and $\mathcal{UC}_i = \mathcal{UC}(\{r_1,\ldots,r_i\})$. We first consider the case where $\overline{K}$ is unbounded (it is possible to construct examples with $m = n$ and $\overline{K} = \Omega(2^n)$). As we add a set to $\mathcal{R}$, the number of atoms can only increase (each atom remains unchanged or is eventually split into two). We thus have $m_i \leq m$ and $|A_i| \leq |\{a \in \mathcal{A}(\mathcal{R}) \mid a \subseteq r_i\}|$. Using $|\mathcal{UC}_i(a)| \leq m_{i+1} \leq m$, the overall complexity is $O(nm + m\log m\sum_i\sum_{a\in\mathcal{A}(r)}|\{r\in\mathcal{R}\mid a\subseteq r\}|) = O(nm + \overline{k}m^2\log m)$ by definition of average overlap. The $O(nm^2)$ bound is obtained by using Algorithm 6.1 instead of Algorithm 6.2.

We now derive a bound depending on the average overlapping degree $\overline{K}$ of combinations. Consider an atom $a \in A_i$ and an uncovered combination $c \in \mathcal{UC}_i(a)$. We can associate a to an atom $a.desc \subseteq a$ in $\mathcal{A}(\mathcal{R})$. As c is also a combination in $\mathcal{C}(\mathcal{R})$, we have $c \in \mathcal{C}(a.desc)$ where $\mathcal{C}(s)$ denotes the combinations of $\mathcal{R}$ containing s. As the atoms in A_i are disjoint, the atoms $a.desc$ for $a \in A_R$ are pairwise distinct. We thus have $\sum_{a\in A_i}|\mathcal{UC}_i(a)| \leq \sum_{a\in\mathcal{A}(\mathcal{R})|a\subseteq r_i}|\mathcal{C}(a)|$. The overall complexity of atom computation is $O(nm + \log m\sum_{a\in\mathcal{A}(\mathcal{R})}\sum_{c\in\mathcal{C}(a)}|\{i\mid a\subseteq r_i\}|) = O(nm + k\log m\sum_{a\in\mathcal{A}(\mathcal{R})}|\mathcal{C}(a)|) = O(nm + k\overline{K}m\log m)$ by definition of overlapping degree and average overlapping degree respectively. The refined bound in terms of $T_{set}, T_{coll}(m), T_{inter}(m), T_{updt}(m)$ is obtained similarly.

□

6.3.2 Application to forwarding loop detection

Theorem 6.2 has the following consequences for forwarding loop detection.

Corollary 6.1 *Given a network $\mathcal{N}$ with collection $\mathcal{R}$ of n rule sets with T_ℓ-bounded representation, forwarding loop detection can be performed in $O(T_\ell \min(n + k\overline{K} \log m, nm)m + \overline{k}n_G m)$ time where m is the number of atoms in $\mathcal{A}(\mathcal{R})$, k is the overlapping degree of $\mathcal{R}$ and $\overline{k}$ (resp. $\overline{K}$) is the average overlapping degree of $\mathcal{R}$ (resp. $\mathcal{C}(\mathcal{R})$). If sets are represented by ℓ-wildcard expressions (resp. (d, ℓ)-multi-ranges), the representation can be computed in $O(\ell \min(n + k\overline{K}, nm)m + \overline{k}n_G m)$ (resp. $O(\ell \min(\overline{k} \log^d m + k\overline{K}, nm)m + \overline{k}n_G m)$) time.*

Corollary 6.1 directly follows from the following claim and Theorem 6.2.

Claim 6.1 *Given the collection $\mathcal{R}$ of rule sets of a network $\mathcal{N}$, and for each atom $a \in \mathcal{A}(R)$ the list $\mathcal{R}(a)$ of sets in $\mathcal{R}$ that contain a, forwarding loop detection can be solved in $O(\overline{k}n_G m)$ time where $m = |\mathcal{A}(\mathcal{R})|$ is the number of header classes, $\overline{k}$ is the average overlapping degree of $\mathcal{R}$ and n_G is the number of nodes in $\mathcal{N}$.*

Proof We assume that each rule set $r \in \mathcal{R}$ is associated with the list L_r of forwarding rules (r, a) that have rule set r. Each such rule is also supposed to be associated to the node u whose table contains it and the index i of the rule in $T(u)$. Each list L_s (cf. Section 6.1.4) is additionally supposed to be sorted according to associated nodes. Such lists can easily be obtained by sorting the collection of all forwarding tables according to the predicate filters of rules.

The claim comes from the fact that uncovered combination in $\mathcal{UC}(\mathcal{R})$ inclusion-wise represent atoms of $\mathcal{A}(\mathcal{R})$. It follows from testing for each header class $a \in \mathcal{A}(\mathcal{R})$ whether the graph $G_a = G_h$ for all $h \in a$ has a directed cycle. G_a is computed by merging the lists L_s for $s \in \mathcal{R}(a)$ in time $O(|\mathcal{R}(a)|\, n_G)$. This graph has at most n_G edges and cycle detection can be performed in $O(n_G)$ time. The overall complexity follows from $\overline{k}m = \sum_{a \in \mathcal{A}(\mathcal{R})} |\mathcal{R}(a)|$ by definition of $\overline{k}$. □

The upper-bounds for forwarding loop detection listed in Table 6.1 follow from Corollary 6.1. A key ingredient consists in maintaining for each rule set a list that describes its presence, priority and effect for each node. Detecting a loop for a header class a then consists in merging the lists associated to the rule sets containing a (as provided by our atom representation) for obtaining the forwarding graph $G_a = G_h$ for all $h \in a$. Directed cycle detection is finally performed on each such graph.

6.4 Theoretical comparison with related work

Previous works on network verification has led to a series of methods for network analysis, resulting in several tools [KZZ+13, KCZ+13, MKA+11, ZZY+14]. The main approaches rely

on computing classes of headers by combining rule predicate filters using intersection and set difference (that is intersection with complement). The idea of considering all header classes generated by the global collection of the sets associated to all forwarding rules in the network is due to Veriflow [KZZ+13]. However, the use of set differences results in computing a refined partition of the atoms of the field of sets generated by this collection that can be much larger than an exact representation. NetPlumber [KCZ+13], which relies on the header space analysis introduced in [KVM12], refines this approach by considering the set of headers that can follow a given path of the network topology. This set is represented as a union of classes that match some rules (those that indicate to forward along the path) and not some others (those that have higher priority and deviate from the path): a similar problem of atom representation thus arises. The idea of avoiding complement operations is somehow approached in the optimization called "lazy subtraction" that consists in delaying as much as possible the computation of set differences. However, when a loop is detected, formal expressions with set differences have to be tested for emptiness. They are then actually developed, possibly resulting in the manipulation of expressions with exponentially many terms.

Concerning the tractability of the problem, the authors of NetPlumber observe a phenomenon called "linear fragmentation" [KVM12] that allows to argue for the efficiency of the method. They introduce a parameter c measuring this linear fragmentation and claim a polynomial time bound for loop detection for low c [KVM12] (when emptiness tests are not included in the analysis). However, the rigorous analysis provided in [KVM11] includes a c^{D_G} factor where D_G is the diameter of the network graph. While this factor appears to be largely overestimated in practice, the sole hypothesis of linear fragmentation does not suffice for explaining tractability and prove polynomial time guarantees. The alternative approach of Veriflow is specifically optimized for rules resulting from range matching within each field of the header. When the number of fields is constant, polynomial time execution can be guaranteed but this result does not extend to general wildcard matching.

A similar problem consists in conflict detection between rules and their resolution [ASP00, EM01, BC15]. It has mainly been studied in the context of multi-range rules [ASP00, EM01], which can benefit from computational geometry algorithms. (A multi-range can be seen as a hyperrectangle in a d-dimensional euclidean space where d is the number of fields composing headers.) Another similar problem, determining efficiently the rule that applies to a given packet, has been extensively studied for multi-ranges [EM01, FM00, GM01]. In the case of wildcard matching, such problems are related to the old problem of partial matching [Riv76]. It is believed to suffer from the "curse of dimensionality" [BOR99, Pat11] and no method significantly faster than exhaustive search is expected to be found with near linear memory (although some tradeoffs are known for small number of $*$ letters [CIP02]). However, efficient hardware based implementation exist based on Ternary Content Addressable Memory (TCAMs) [BGK+13] or

Graphics Processing Unit (GPU) [VLZL14].

6.4.1 Related notion of weak completeness

Most of the previous work rely on complement computations (or equivalent operations): the complement of a single set generally requires to be represented as a union of several intermediate sets (up to ℓ for ℓ-wildcards and up to $2d-1$ for d-multi-ranges). We now provide examples where this can lead to exponential blow-up. The notion of uncovered combination is linked to that of weak completion introduced by [BC15] in the context of rule-conflict resolution as detailed in this section. In the context of resolution of conflicts between rules, Boutier and Chroboczek [BC15] introduce the concept of *weak completeness*: a collection $\mathcal{R}$ is weakly complete iff for any sets $r, r' \in \mathcal{R}$, we have $r \cap r' = \cup_{r'' \subseteq r \cap r'} r''$. They show that this is a minimal necessary and sufficient condition for all rule conflicts to be solved when priority of rules extends inclusion (i.e. r has priority over r' when $r \subsetneq r'$). Interestingly, we can make the following connection with this work: given a combination collection $\mathcal{C}' \subseteq \mathcal{C}(\mathcal{R})$ containing $\mathcal{UC}(\mathcal{R})$, we have $a(c) = c \setminus \cup_{c' \in \mathcal{C}' | c' \subsetneq c} c'$ for all $c \in \mathcal{C}'$. (See Lemma 6.2 in Section 6.3.1). This allows to show that $\mathcal{UC}(\mathcal{R})$ is weakly complete. It is indeed the smallest collection of combinations of $\mathcal{R}$ that contains $\mathcal{R} \cup \{H\}$ and that is weakly complete. Our work thus also provides an algorithm for computing such an optimal "weak completion".

6.4.2 Lower bound for HSA / NetPlumber

HSA/NetPlumber [KVM12, KCZ+13] use clever heuristics to efficiently compute the set of headers H_P than can traverse a given path P. An important one consists in lazy subtraction: set difference computations are postponed until the end of the path. For that purpose, this set H_P is represented as a union of terms of the form $s = c_0 \setminus \cup_{i=1..p} c_i$ where the elementary sets $c_0, \ldots, c_p$ are represented with wildcards. The emptiness of such terms is regularly tested. A simple heuristic is used during the construction of the path: s is obviously empty if c_0 is included in c_i for some $i \geq 1$. But if the path loops, HSA has to develop the corresponding terms into a union of wildcards to determine if one of them may produce a forwarding loop.

We now provide an example where this emptiness test can take exponential time. Consider a node whose forwarding table consists in $\ell + 1$ rules with following rule sets:

$$r_0 = 1^\ell, \qquad r_i = 1^{\ell-i} 0 *^{i-1} \quad for \quad i = 1..\ell, \qquad r_{\ell+1} = *^\ell. \tag{6.3}$$

All rules are associated with the drop action except the last rule (with rule set $r_{\ell+1}$) whose action is to forward to the node itself. Such a forwarding table is depicted in Figure 5.1 for $\ell = 4$. Starting a loop detection from that node, HSA detects a loop for headers in $r_{\ell+1} \setminus \cup_{i=0..\ell} r_i$. The emptiness of this term is thus tested. For that purpose, HSA represents the complement of r_i with $0*^{\ell-1} \cup *0*^{\ell-2} \cup \cdots \cup *^{\ell-i-1}0*^{i} \cup *^{\ell-i}1*^{i-1}$. Note that each of the $\ell - i$ wildcard expressions in that union have only one non-$*$ letter. Distributivity is then used to compute $r_{\ell+1} \setminus \cup_{i=0..\ell} r_i$ as $\overline{r_0} \cap \cdots \cap \overline{r_\ell}$. After expanding the first $j-1$ intersections, HSA thus obtains a union of wildcards with j letters in $\{0,1\}$ and $\ell - j$ letters equal to $*$ that has to be intersected with $\overline{r_{j+1}} \cap \cdots \cap \overline{r_\ell}$. In particular, this unions contains all strings with j letters equal to 0 and $\ell - j$ equal to $*$. All ℓ-letter strings with alphabet $\{*, 0\}$ are produced during the computation which thus requires $\Omega(\ell 2^\ell)$ time. For testing a network with n_G similar nodes, HSA thus requires time $\Omega(\ell n_G 2^\ell)$. As all sets $r_0, \ldots, r_\ell$ are pairwise disjoint, the overlapping degree of the collection is $k_{\max} = 2$ and this justifies the two lower-bounds indicated for NetPlumber in Table 6.1.

The NetPlumber approach could be generalized to more general types of rules. However, we show that the simple heuristic for emptiness tests is not sufficient. We provide an example where the HSA/NetPlumber approach generates an exponential number of paths while the number of classes is linear if it relies solely on this heuristic. Consider header space $H = \{1..n\}$ and the following $n+1$ rule sets:

$$r_1 = \overline{\{1\}}, \quad \ldots, \quad r_n = \overline{\{n\}} \qquad \textit{and} \quad r_{n+1} = H \tag{6.4}$$

Consider a network $\mathcal{N}$ with $n_G = n(n+1)$ nodes. Each node $u_{i,j}$ for $0 \leq i \leq n$ and $1 \leq j \leq n$ has table $T(u_{i,j}) = (r_{i+1}, FwdD_{i+1,1}), \ldots, (r_n, FwdD_{i+1,n}), (r_{n+1}, FwdD_{i+1,n})$ where action $FwdD_{i,j}$ indicates to forward packets to node $u_{i,j}$ for $i \leq n$ and to drop packets for $i = n+1$. Starting from $u_{0,1}$, the HSA approach generates a path for each combination $r_{i_1} \cap \cdots \cap r_{i_p}$ for $p \leq n$ and $1 \leq i_1 < \cdots < i_p \leq n$. This path goes through $u_{0,1}, u_{1,i_1}, \ldots, u_{p,i_p}$ and then through $u_{p+1,n}, \ldots, u_{n,n}$. It is constructed at least for term $r_{i_1} \cap \cdots \cap r_{i_p} \setminus \cup_{j \notin \{i_1,\ldots,i_p\}} r_j$. The heuristical emptiness test of NetPlumber does not detect that it is empty since $r_{i_1} \cap \cdots \cap r_{i_p}$ contains j for $j \notin \{i_1, \ldots, i_p\}$ and it is not included in r_j. The number of paths generated is thus at least $\sum_{1 \leq p \leq n} \binom{n}{p} = 2^n - 1$. However, the header classes are all singletons of H and their number is $m = n$. Note the high overlapping degree $k_{\max} = n$ of this collection of rule sets.

6.4.3 Lower bound for VeriFlow

VeriFlow [KZZ+13] incrementally computes a partition into sub-classes that forms a refinement of the header classes: when a rule r is added, each sub-class c is replaced by $c \cap r$ and a partition

of $c \setminus r$. Veriflow benefits from the hypothesis that headers can be decomposed into d fixed fields and that each rule set can be represented by a multi-range $r = [a_1, b_1] \times \cdots \times [a_d, b_d]$. The intersection of two multi-ranges is obviously a multi-range. However, set difference is obtained by intersection with the complement which is represented as the union of up to $2d-1$ multi-ranges.

The complementary of a multi-range $r = [a_1, b_1] \times \cdots \times [a_d, b_d]$ is represented as the union of $2d-1$ multi-ranges (at most):

$$
\begin{array}{l}
[0, a_1 - 1] \times H_{2..d} \text{ and } [b_1 + 1, \infty_1] \times H_{2..d}, \\
[a_1, b_1] \times [0, a_2 - 1] \times H_{3..d} \text{ and } [a_1, b_1] \times [b_2 + 1, \infty_2] \times H_{3..d}, \\
\cdots, \\
[a_1, b_1] \times \cdots \times [a_{d-1}, b_{d-1}] \times [0, a_d - 1] \text{ and } [a_1, b_1] \times \cdots \times [a_{d-1}, b_{d-1}] \times [b_d + 1, \infty_d],
\end{array}
$$

where ∞_i denotes the maximum possible value in field i, and $H_{i..j} = [0, \infty_i] \times \cdots \times [0, \infty_j]$ denotes the multi-range of all possible values for fields $i, \ldots, j$ for $1 \leq i \leq j \leq d$.

The difficult input for Veriflow consists in a network with $n = dp + 1$ rules associated to the following multi-ranges:

$$
r_0 = H_{1..d}, \qquad r_i^j = H_{1..i-1} \times [a_j, a_j] \times [b, b]^{d-i} \qquad for \quad i, j \in [1..d] \times [1..p] \tag{6.5}
$$

Consider the sub-classes generated while computing $r_0 \cap \left(\cap_{i,j \in [1..d] \times [1..p]} \overline{r_i^j} \right)$. The union of multi-ranges representing $\overline{r_i^j}$ contains in particular $H_{1..i-1} \times [0, a_j - 1] \times H_{i+1..d}$ and $H_{1..i-1} \times [a_j + 1, \infty_i] \times H_{i+1..d}$. This implies that Veriflow generates on such an input all p^d sub-classes of the form $I_1 \times \cdots \times I_d$ with $I_i = [a_j + 1, a_{j+1} - 1]$ for some $j \in [0, d]$ (we set $a_0 = -1$). Forwarding loop detection of an n_G-node network thus requires $\Omega(p^d n_G) = \Omega(\left(\frac{n}{d}\right)^d n_G \frac{m}{d})$ time for Veriflow. As this example has overlapping degree 2, this justifies the two lower-bounds indicated for Veriflow in Table 6.1 for d-multi-ranges.

It is possible to adapt Veriflow to support general wildcard matching by considering each field bit as a field. The wildcard expressions $r_0 = *^\ell, r_1 = 01^{\ell-1}, \ldots, r_\ell = 0^{\ell-1}1$ will then similarly generate all $2^{\ell/2}$ sub-classes obtained by concatenation of words 10 and 11. This justifies the two lower-bounds indicated for Veriflow in Table 6.1 for ℓ-wildcards.

6.4.4 Linear fragmentation versus overlapping degree

Interestingly, a complexity analysis of HSA loop detection is given in the technical report [KVM11] under an assumption called "linear fragmentation". This assumption, which is based on empirical observations, basically states that there exists a constant c such that a given rule set intersects at most c of the terms of the set H_P generated by the rules along a given path P in the graph of the network. One can then easily prove by induction that the number of terms generated by the rules along a path of length p is $c^p n$ at most. Under linear-fragmentation, the time complexity of HSA loop detection (excluding emptiness tests) is thus proved to be $O(c^{D_G} D_G n^2 m_G)$ in [KVM11] where D_G is the diameter of network graph G, n the number of rules, and m_G the number of ports in G (in our simplified model each node has a single input port and $m_G = n_G$ the number of nodes in G). It is then argued that in practice the constant c gets smaller as the length p of the path considered increases and that practical loop detection has complexity $O(D_G n^2 m_G)$ as claimed in [KVM12]. However, it is not rigorous to neglect the (exponential) c^{D_G} factor under the sole linear-fragmentation hypothesis.

Additionally, we think that low overlapping degree provides a simple explanation for the phenomenon observed by Kazemian et al.: as the path length increases, the terms representing the header that can traverse the path result from the intersection of more rules and become less likely to intersect other rules when overlapping degree is limited. Moreover, bounded overlapping degree $k_{\max}$ implies that the number of paths and terms generated by HSA is bounded by $O(n_G n^{k_{\max}})$. This guarantees that all HSA computations besides emptiness tests remain polynomial for constant $k_{\max}$. In contrast, we provide in Section 6.4.2 an example with unbounded overlapping degree where the HSA approach can generate exponentially many paths compared to the number of header classes in the context of general rules.

6.5 Conclusion

We conclude this chapter summarizing the most important results.

We focused on the problem of finding forwarding loops in a given network instance. Our problem is defined as following: given a network topology and all nodes' forwarding tables, are we able to detect if there exists at least one packet header s.t. a packet is continuously forwarded by network nodes in a cycle? We adopted the approach of checking routers' forwarding rules to verify if they can create loops for some packet headers.

Keeping in mind that forwarding rules validation is known to be an NP-complete problem, we focused on reducing the number of tests to be performed proposing a novel representation of

network forwarding classes. In forwarding networks, each forwarding rule can be associated with the set of packet headers it matches. We showed that we can perform the verification task on the network equivalence classes that correspond to the headers that match exactly the same rules. These classes, called *atoms*, are intuitively the set of headers that share the same forwarding behavior in the network.

We canonically represent the atoms by means of uncovered combinations of forwarding rules w.r.t. set intersection. This allows us to ignore the complement set operation, which can increase the complexity of representing the network classes even in simple cases such as range rule (cf. Figure 6.1).

We proposed an efficient algorithm to incrementally compute the atoms, and take advantage of a network "measure" that we defined: the overlapping degree. When the intersection of two sets can be efficiently computed and represented, our algorithm is polynomial in the number of header classes. Additionally, we showed that the overlapping degree in real networks is very low: we can bound the overlapping degree by a constant, which furtherly ensures polynomial number of header classes.

Our framework can be used to detect not only forwarding loops, but also other classical problems:

- finding all black-holes (e.g. detecting whether there is a node in the network where packets are incorrectly dropped);
- detect reachability between two points (e.g. verifying if there exists at least one header packet that connects a source and a destination node).

Finally, we believe that our tool can replace the canonical representation of existing verification tools based on network classes, in order to speed-up the problem detection thanks to our better representation.

Table of symbols, part II

Network modeling	
$\mathcal{D}$	A generic data structure
$G = (V, E)$	Network graph, ov V nodes and E edges
h	A generic header
H	The header space, with $h \in H$
l	Number of bits of a specific header
n_G	Number of nodes in the network graph
n	Number of rules, or size of the rule set
$\mathbf{P}$	Anteater policies, representing the forwarding actions
r	A generic rule of the rule set, $r \in \mathcal{R}$
$\mathcal{R}$	Collection of the forwarding rules
s	Element stored in a data structure
u, v	Generic nodes of the network $u, v \in V$. uv is an edge, or $uv \in E$
Rules, atoms and combinations	
$\mathcal{A}(\mathcal{R})$	The set of network classes, or the atoms of $\mathcal{R}$.
$\mathcal{C}(\mathcal{R})$	The set of rule combinations, or the set of all c for $\{c = \cap_{r \in \mathcal{R}} r\}$
c	A generic combination of rules, $c = r_1 \cap \ldots \cap r_k$, for $k \leq n$
m	Number of classes, or size of the atoms set
$T(u)$	Forwarding table of node $u \in V$
Network parameters and complexity analysis	
D_G	Diameter of the network graph
d	Number of fields in a multirange rule
k	Overlapping degree of $\mathcal{R}$, that is the maximum number of containers.
$\overline{k}$	Average overlapping degree of $\mathcal{R}$
k_{max}	Constant value which bounds the overlapping degree of the network
$\overline{K}$	Average overlapping degree of the combinations $\mathcal{C}(\mathcal{R})$
T_H	Time boundary of set intersection or cardinality computation in H
T_l	" in a wildcard l-bit space

Conclusion

Conclusion

The Internet usage is strongly affected by the diffusion of new services causing different paradigms to quickly emerge. On the contrary, the underlying network infrastructure cannot be upgraded at a comparable speed: the Internet evolution is indeed a challenge. In this thesis we followed two main research directions: first, we focused on designing, prototyping and evaluating Caesar, a device which is capable of introducing content-based functionalities to real commodity network equipment, being fully compatible with current networks; second, we developed a mathematical framework to analyze the problem of verifying SDN networks to check the presence of network loops, thus resulting in a software tool called IHC, that can check the existence of loops in real networks. We now summarize the achievements of this thesis work.

Summary of Thesis Achievements

In the first part of the thesis we focused on the design, implementation and performance evaluation of a content router, called Caesar, which is capable of performing content-based operations in network packets at high-speed.

Forwarding In chapter 3 we tackled the problem of forwarding packets in NDN, exploiting current network architecture and integrating name-based functionalities on commodity hardware. This chapter shows three important results. First, an algorithm for name-based lookup and forwarding on content-names, which exploits the novel prefix Bloom filter (PBF) data structure to allow efficient longest prefix matching operations. Second, it is fully compatible with current protocols and network equipments. Its design allows network providers to softly upgrade their hardware (i.e. firmware upgrade) in order to exploit the content-based functionalities without the need of redesigning or upgrading the whole network. Third, Caesar can work at high-speed, namely tens of millions packets per second, thus matching existing edge routers for small/medium network sizes.

We performed an extensive experimental evaluation of Caesar's forwarding module, dissecting the bottlenecks and highlighting the feasibility of our design in existing network equipments. We showed that Caesar's performance can match the requirements due to the growth of the forwarding tables or an increasing desired throughput thanks to its modular extension of distributed forwarding and GPU offloading. We managed to forward packets at a rate greater than 6.5 Mpps for a single line card with the reference workload, using content names of $\nu = 42$ bytes, average component distance $d = 2$ and forwarding table of $n = 10M$ elements. Throughput can be furtherly increased in the order of hundreds of Mpps when offloading traffic to an external GPU, and we can increase the FIB size of a factor $\mathcal{N}$ with only a 15% drop.

PIT In chapter 4 we focused on the soft-state management of NDN, which requires that each router store the requests propagated and not yet served. This chapter showed two main results. First, a design of a data-structure which can perform updates (insert, delete, remove) and lookups at high-speed, matching the requirements of the NDN paradigm. Second, the PIT module is easily integrated on the Caesar chassis, being therefore fully compatible with current Internet infrastructure.

We performed as well an extensive experimental evaluation of PIT's performance, showing that a run-time state management is feasible. We are able to continuously perform insert and remove with a rate of 7.9 Mpps in a flow-balanced scenario, performing an exact match on a table of 1M elements and the reference workload.

* * *

In the second part of the thesis we presented an innovative approach for network verification in the SDN environment. We got the inspiration from the related work on Network verification, and targeted the problem of significantly reducing the computation time (i.e. the number of tests to perform) in order to verify a given properties. We theoretically analyzed the problem of detecting all possible loops in a given network, with forwarding rules that may assume the general form of wildcard expressions matching the incoming packet headers.

Rule verification through atoms computation In chapter 6 we targeted the problem of loop-detection in SDN, obtaining the following results. Despite the NP-completeness of this problem, we showed that our approach advances the state-of-the-art verification tools by two main factors: first, its innovative representation of header classes sensibly reduces the input size of the verification problem with respect to the related work; second, the incremental algorithm for the calculation of the header classes (and the subsequent loop detection query) shows an

important speed-up of the state-of-the-art tools. Our model can be generalized for any existing network and other classical network verification problems such as black-holes detection and reachability checks.

We derived a parameter for practical networks, called the overlapping degree of forwarding rules, which allowed us to bound the complexity of the verification process to be polynomial in the number of header classes. Numerical experiments on real datasets (including BGP and firewall traces) showed that the overlapping degree in real environments is low, improving the effectiveness of software verification tools based on header classes computation.

Future Work

We present now some perspectives that could be accomplished as future work.

NDN security and privacy Security and privacy issues are not taken into account for this thesis. However, they cannot be traditionally investigated due to the significant difference between current Internet techniques and NDN; moreover, the coexistence with current network infrastructure may require to differentiate the actual implementation depending on the traffic pattern. We plan to address one typology of Denial of Service (DoS) attack, namely the *Interest flooding* [AMM+13]. It consists in the pollution of the PIT table with requests for unknown or very unpopular content, causing the PIT to fail the Data transmission back to the users. As proposed in [AMM+13] a possible solution could be to mark the malicious Interest packets: such packets are in fact almost never matched by incoming Data packets, and are removed as soon as the corresponding timer expires.

While users' identity is usually hidden in the NDN architecture, when access control is required (i.e. firewall, corporate proxies), some novel authentication mechanisms may be needed in order to be granted with the requested permissions without losing anonymity. Authors of [LLR+12] showed that caching may affect users' privacy because it is possible to estimate (via cache probing) the objects locally cached to gather privacy-sensitive information. A proposed countermeasure could be to avoid caching of privacy-sensitive objects, because they show a low (local) popularity and caching does not increase network performance. We plan to design a Content Store whose caching strategy is privacy-safe.

NDN control plane The control plane development is still a challenge in NDN. As part of the control plane, existing routing protocols may fail in managing the number of updates when

lots of content replicas are disseminated to the network nodes. In fact, the original NDN proposal, considers only a basic routing protocol to advertise contents on original servers [JSB+09], while more recent works propose to advertise the presence of the replicas through the control plane [WLV+12], though admitting that the solution is not much scalable to support all the Internet contents.

Furthermore, the population of forwarding tables may be a difficult task because of the size of the address space, and the volatile property of the route advertisements, which may change suddenly when cached advertised contents are evicted.

* * *

Rule verification with write action In this thesis we did not take into account the "write" rules, which are rules defining partial header modifications. Our framework can be easily adapted for particular cases of writes, such as MPLS [DR00], where write actions consist of adding, removing or modifying a integer label at the end of the packet header. Generic writes may modify the header space generating several additional classes. Moreover, header modifications may translate in new forwarding decisions, thus resulting in a more complex forwarding graph.

However, we believe that our framework proposed in Chapter 6 could be integrated in NetPlumber [KCZ+13], which support the write actions. Updating a collection of uncovered combinations can be done in persistent style for managing efficiently several collections as they follow different paths. Write operations could be recorded in a specific wildcard expression that serves as a general mask for all wildcards in the collection, allowing to apply efficiently such "network transfer function" (using HSA/NetPlumber terminology) to a collection. This would allow to enhance the emptiness tests performed within NetPlumber to guarantee polynomial time execution when the number of header classes is polynomially bounded.

VMs and multi-level network verification Existing work in SDN verification tools focuses on network layer, mostly considering forwarding rules. As described in Section 1.5.3 at page 31, SDN may be exploited to implement and simplify existing virtualization mechanisms. We believe that our framework may be extended to support more advanced high-level verification policies. We provide a few simple example of this direction. Consider a SDN-network implementing several virtual networks sharing the same physical infrastructure. Every network may be granted with different bandwidth provisioning, possibly resulting in some virtual network having less resources than required. It may be possible to exploit our class representation

to detect that packets of that particular virtual network are not able to go out of the local scope even if forwarding rules are correct and the whole network is loop-free. In addition, when several virtualization levels are present, verification tasks may be challenging due to the increase of network classes (similarly to the "write" scenario).

Performance evaluation of IHC and comparison with related work We developed a preliminary version of a software verification tool based on IHC, which has been developed with high-level languages. We preliminary tested the IHC's algorithm for atom computation on a Linux laptop, showing that a complex network task such as network verification can be performed even on commodity PCs.

However, in order to take advantage of IHC's representation a more performing language is required: we plan to develop a C/C++ plugin and we believe that our approach could be easily integrated in the Veriflow [KZZ+13] core library both for speed-up and performance-guarantee considerations. Our idea is to replace VeriFlow's class computation module, which can generate a great number of sub-classes (and therefore a higher number of elements must be checked to detect loops) with our class representation: we expect to observe a performance speed-up thanks to our smaller number of atoms.

Finally, we are interested in empirically evaluating the feasibility of IHC deployment in a real network environment, and comparing our performance results with the state-of-the-art tools for network diagnosis, namely VeriFlow and NetPlumber/HSA.

Glossary

ARP Address Resolution Protocol. It is used to match IP addresses to corresponding MAC addresses [Plu82].

BGP Border gateway protocol. It is a protocol for the decision of routes among different *autonomous systems*. Every autonomous system is an independent network, and routes are not decided through a shortest-path algorithm, but rather by means of service level agreements among network providers [RL95].

DNS Domain name system [Pos94]. It is a hierarchical decentralized database mapping URLs (dot-delimited human-readable strings such as `www.inria.fr`) to effective 32-bit IP addresses.

ICMP The Internet Control Message Protocol's purpose is to provide feedback about problems in the network. ICMP is used for example, to report an error in datagram processing [Pos81].

LC Abbreviation of line card.

MPEG Moving Picture Experts Group (MPEG) is a standard which defines the proper coding for media type such as audio and video.

OSPF Acronym for Open Shortest Path First. It is a routing protocol, based on the Dijkstra algorithm, that responds quickly to topology changes, yet involves small amounts of routing protocol traffic [Moy97].

RFID Radio-frequency identification (RFID), it is a technology that exploits electromagnetic fields to tag objects. An example of RFIDs objects is a corporate badge.

RIP Acronym for Routing Information Protocol. This routing protocol, based on the Bellman-Ford algorithm, has been used for routing computations in computer networks since the early days of the ARPANET [Hed88].

SSL Secure socket layer. It encrypts a channel between two endpoints to create a secure and reliable connection.

TCP Transport Control Protocol. It implements a connection-oriented reliable channel between endpoints over an unreliable network.

UDP User Datagram protocol. The most simple protocol which can provide a best-effort delivery channel between endpoints.

Bibliography

[ABDDP13] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè, *Cloud monitoring: A survey*, Computer Networks **57** (2013), no. 9, 2093–2115.

[AD11] Saamer Akhshabi and Constantine Dovrolis, *The evolution of layered protocol stacks leads to an hourglass-shaped architecture*, ACM SIGCOMM Computer Communication Review **41** (2011), no. 4, 206.

[AD12] Bengt Ahlgren and Christian Dannewitz, *A survey of information-centric networking*, Communication Magazine, IEEE **50** (2012), no. July, 26–36.

[AMM+13] Alexander Afanasyev, Priya Mahadevan, Ilya Moiseenko, Ersin Uzun, and Lixia Zhang, *Interest flooding attack and countermeasures in named data networking*, IFIP Networking Conference, 2013, IEEE, 2013, pp. 1–9.

[ASP00] Hari Adiseshu, Subhash Suri, and Guru M. Parulkar, *Detecting and resolving packet filter conflicts*, Proceedings IEEE INFOCOM 2000, The Conference on Computer Communications, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Reaching the Promised Land of Communications, Tel Aviv, Israel, March 26-30, 2000, IEEE, 2000, pp. 1203–1212.

[AYW+] Alexander Afanasyev, Cheng Yi, Lan Wang, Beichuan Zhang, and Lixia Zhang, *Map-and-encap for scaling ndn routing*, Tech. report.

[Bar12] MF Bari, *A survey of naming and routing in information-centric networks*, Communication Magazine, IEEE **50** (2012), no. December.

[BC15] Matthieu Boutier and Juliusz Chroboczek, *Source-specific routing*, IFIP Networking, 2015.

[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker, *Web caching and zipf-like distributions: Evidence and implications*, INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, vol. 1, IEEE, 1999, pp. 126–134.

[BCKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars, *Computational geometry: Algorithms and applications*, 3rd ed. ed., Springer-Verlag TELOS, Santa Clara, CA, USA, 2008.

[BDLPL+15] Yacine Boufkhad, Ricardo De La Paz, Leonardo Linguaglossa, Fabien Mathieu, Diego Perino, and Laurent Viennot, *Vérification de tables de routage par utilisation d'un ensemble représentatif d'en-têtes*, ALGOTEL 2015 - 17èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications (Beaune, France), June 2015.

[Bel56] Richard Bellman, *On a routing problem*, Tech. report, DTIC Document, 1956.

[BGK+13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz, *Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN*, Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (New York, NY, USA), SIGCOMM '13, ACM, 2013, pp. 99–110.

[Blo70] Burton H Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM **13** (1970), no. 7, 422–426.

[BM99] Scott Bradner and Jim McQuaid, *Benchmarking Methodology for Network Interconnect Devices*, RFC 2544, 1999.

[BM04] Andrei Broder and Michael Mitzenmacher, *Network applications of bloom filters: A survey*, Internet mathematics **1** (2004), no. 4, 485–509.

[BOR99] Allan Borodin, Rafail Ostrovsky, and Yuval Rabani, *Lower bounds for high dimensional nearest neighbor search and related problems*, Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA (Jeffrey Scott Vitter, Lawrence L. Larmore, and Frank Thomson Leighton, eds.), ACM, 1999, pp. 312–321.

[C+12] M Chiosi et al., *Network functions virtualisation–introductory white paper*, SDN and OpenFlow World Congress, Darmstadt, Germany, 2012.

[CB10] NM Mosharaf Kabir Chowdhury and Raouf Boutaba, *A survey of network virtualization*, Computer Networks **54** (2010), no. 5, 862–876.

[Cen] Palo Alto Research Center, a CCNx software implementation, `http://blogs.parc.com/ccnx/`.

[CGM12] G. Carofiglio, M. Gallo, and L. Muscariello, *Icp: Design and evaluation of an interest control protocol for content-centric networking*, Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on, March 2012, pp. 304–309.

[CIP02] Moses Charikar, Piotr Indyk, and Rina Panigrahy, *New algorithms for subset query, partial match, orthogonal range searching, and related problems*, Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings (Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, eds.), Lecture Notes in Computer Science, vol. 2380, Springer, 2002, pp. 451–462.

[Cis] Cisco, http://www.ciscopress.com/articles/article.asp?p=174313&seqNum=5.

[CK74] V. Cerf and R. Kahn, *A Protocol for Packet Network Intercommunication*, Toc **22** (1974), no. 5.

[CPW11] Antonio Carzaniga, Michele Papalini, and Alexander L Wolf, *Content-Based Publish / Subscribe Networking and Information-Centric Networking*, ICN, 2011, pp. 56–61.

[DEA+09] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy, *Routebricks: exploiting parallelism to scale software routers*, SOSP '09 (Big Sky, Montana, USA), 2009.

[DHM+13] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella, *Towards an elastic distributed sdn controller*, ACM SIGCOMM Computer Communication Review **43** (2013), no. 4, 7–12.

[Dij59] Edsger W Dijkstra, *A note on two problems in connexion with graphs*, Numerische mathematik **1** (1959), no. 1, 269–271.

[DLCW12] Huichen Dai, Bin Liu, Yan Chen, and Yi Wang, *On pending interest table in named data networking*, Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems - ANCS '12 (2012), 211.

[DLWL15] Huichen Dai, Jianyuan Lu, Yi Wang, and Bin Liu, *Bfast: Unified and scalable index for ndn forwarding architecture*, Computer Communications (INFOCOM), 2015 IEEE Conference on, IEEE, 2015, pp. 2290–2298.

[DR00] Bruce Davie and Yakov Rekhter, *Mpls: technology and applications*, Morgan Kaufmann Publishers Inc., 2000.

[EM81] Herbert Edelsbrunner and Hermann A. Maurer, *On the intersection of orthogonal objects*, Information Processing Letters **13** (1981), no. 4, 177–181.

[EM01] David Eppstein and S. Muthukrishnan, *Internet packet filter management and rectangle geometry*, Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA. (S. Rao Kosaraju, ed.), ACM/SIAM, 2001, pp. 827–835.

[EMP+82] Herbert Edelsbrunner, Hermann A. Maurer, Franco P. Preparata, Arnold L. Rosenberg, Emo Welzl, and Derick Wood, *Stabbing line segments*, BIT Numerical Mathematics **22** (1982), no. 3, 274–281.

[FAK13] Bin Fan, David G. Andersen, and Michael Kaminsky, *Memc3: Compact and concurrent memcache with dumber caching and smarter hashing*, Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13) (Lombard, IL), USENIX, 2013, pp. 371–384.

[FCAB98] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder, *Summary cache: A scalable wide-area web cache sharing protocol*, ACM SIGCOMM Computer Communication Review, vol. 28, ACM, 1998, pp. 254–265.

[FGT92] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier, *Birthday paradox, coupon collectors, caching algorithms and self-organizing search*, Discrete Applied Mathematics **39** (1992), no. 3, 207–229.

[FLYV93] Vince Fuller, Tony Li, Jessica Yu, and Kannan Varadhan, *Classless inter-domain routing (cidr): an address assignment and aggregation strategy.*

[FM00] Anja Feldmann and S. Muthukrishnan, *Tradeoffs for packet classification*, Proceedings IEEE INFOCOM 2000, The Conference on Computer Communications, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Reaching the Promised Land of Communications, Tel Aviv, Israel, March 26-30, 2000, IEEE, 2000, pp. 1193–1202.

[For56] Lester Randolph Ford, *Network flow theory.*

[GGM12] Massimo Gallo, Carofiglio Giovanna, and Luca Muscariello, *Joint Hop-by-hop and Receiver-Driven Interest Control Protocol for Content-Centric Networks*, ACM SIGCOMM ICN, 2012 (Helsinki, Finland), August 2012.

[GJN11] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan, *Understanding network failures in data centers: measurement, analysis, and implications*, ACM SIGCOMM Computer Communication Review, vol. 41, ACM, 2011, pp. 350–361.

[GM01] Pankaj Gupta and Nick McKeown, *Algorithms for packet classification*, IEEE Network: The Magazine of Global Internetworking **15** (2001), no. 2, 24–32.

[HAA+13] A K M Mahmudul Hoque, Syed Obaid Amin, Adam Alyyan, Beichuan Zhang, Lixia Zhang, and Lan Wang, *NLSR: Named-data Link State Routing Protocol*, Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking - ICN '13 (2013), 15.

[Hed88] Charles L Hedrick, *Routing information protocol version 2*, RFC 2453, 1988.

[HGJL15] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, *Network function virtualization: Challenges and opportunities for innovations*, IEEE Communications Magazine **53** (2015), no. 2, 90–97.

[icn] *Information centric networking research group (icnrg)*, `http://irtf.org/icnrg`.

[IM03] Sundar Iyer and Nick W. McKeown, *Analysis of the parallel packet switch architecture*, IEEE/ACM Trans. Netw. **11** (2003), no. 2, 314–324.

[Ind14] Cisco Visual Networking Index, *Cisco Visual Networking Index: Forecast and Methodology, 2014 – 2019*, White Paper (2014).

[Ind15] ———, *Global mobile data traffic forecast update, 2010-2015*, White Paper (2015).

[Int] DPDK Intel, *Data plane development kit*, URL http://dpdk. org.

[JCDK01] Kirk L. Johnson, John F. Carr, Mark S. Day, and M. Frans Kaashoek, *The measured performance of content distribution networks*, Computer Communications **24** (2001), no. 2, 202–206.

[JP13] Raj Jain and Sudipta Paul, *Network virtualization and software defined networking for cloud computing: a survey*, Communications Magazine, IEEE **51** (2013), no. 11, 24–31.

[JSB+09] Van Jacobson, Diana K Smetters, Nicholas H Briggs, James D Thornton, Michael F Plass, and Rebecca L Braynard, *Networking Named Content*, Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies CoNEXT '09 (2009), 1–12.

[Kaz] Kazemian Peyman, *HSA/NetPlumber source code repository*, `https://bitbucket.org/peymank/hassel-public/`.

[KC04] P. Koopman and T. Chakravarty, *Cyclic redundancy code (crc) polynomial selection for embedded networks*, Dependable Systems and Networks, 2004 International Conference on, June 2004, pp. 145–154.

[KCZ+13] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte, *Real time network policy checking using header space analysis*, Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), 2013, pp. 99–111.

[KDA12] Vasileios Kotronis, Xenofontas Dimitropoulos, and Bernhard Ager, *Outsourcing the routing control logic: Better internet routing based on SDN principles*, Proceedings of the 11th ACM Workshop on Hot Topics in Networks (New York, NY, USA), HotNets-XI, ACM, 2012, pp. 55–60.

[KMV10] Adam Kirsch, Michael Mitzenmacher, and George Varghese, *Hash-based techniques for high-speed packet processing*, Algorithms for Next Generation Networks, Springer, 2010, pp. 181–218.

[Knu98] Donald Ervin Knuth, *The art of computer programming: sorting and searching*, vol. 3, Pearson Education, 1998.

[KRW13] Naga Praveen Katta, Jennifer Rexford, and David Walker, *Incremental consistent updates*, Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (New York, NY, USA), HotSDN '13, ACM, 2013, pp. 49–54.

[KVM11] Peyman Kazemian, George Varghese, and Nick McKeown, *Header space analysis: Static checking for networks*, Tech. report, Stanford, 2011.

[KVM12] Peyman Kazemian, George Varghese, and Nick McKeown, *Header space analysis: Static checking for networks*, Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), 2012, pp. 113–126.

[KZZ+13] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey, *Veriflow: Verifying network-wide invariants in real time*, Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), 2013, pp. 15–27.

[LBWJ12] Zhaogeng Li, Jun Bi, Sen Wang, and Xiaoke Jiang, *Compression of pending interest table with Bloom filter in content centric network*, Proceedings of the 7th International Conference on Future Internet Technologies - CFI '12 (2012), 46.

[LIJM+11] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian, *Internet inter-domain traffic*, ACM SIGCOMM Computer Communication Review **41** (2011), no. 4, 75–86.

[Lit61] John DC Little, *A proof for the queuing formula: L= λ w*, Operations research **9** (1961), no. 3, 383–387.

[LLR+12] Tobias Lauinger, Nikolaos Laoutaris, Pablo Rodriguez, Thorsten Strufe, Ernst Biersack, and Engin Kirda, *Privacy implications of ubiquitous caching in named data networking architectures*, Tech. report, Technical report, TR-iSecLab-0812-001, iSecLab, 2012.

[LLZ14] Zhuo Li, Kaihua Liu, and Yang Zhao, *MaPIT : An Enhanced Pending Interest Table for NDN with Mapping Bloom Fi*, Communication Letters, IEEE **18** (2014), no. 11, 1915–1918.

[LMEZG97] Hang Liu, Hairuo Ma, Magda El Zarki, and Sanjay Gupta, *Error control schemes for networks: An overview*, Mob. Netw. Appl. **2** (1997), no. 2, 167–182.

[Ltd13] Point Topic Ltd, *VoIP statistics - market analysis*, `http://point-topic.com/wp-content/uploads/2013/02/Point-Topic-Global-VoIP-Statistics-Q1-2013.pdf`, 2013.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner, *OpenFlow:Enabling Innovation in Campus Networks*, ACM SIGCOMM Computer Communication Review **38** (2008), no. 2, 69.

[MCG11] Luca Muscariello, Giovanna Carofiglio, and Massimo Gallo, *Bandwidth and storage sharing performance in information centric networking*, Proceedings of the ACM SIGCOMM workshop on Information-centric networking, ACM, 2011, pp. 26–31.

[MKA+11] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King, *Debugging the data plane with anteater*, ACM SIGCOMM Computer Communication Review **41** (2011), no. 4, 290–301.

[Moy97] John Moy, *OSPF version 2*, RFC 2328, 1997.

[MPIA09] Gregor Maier, Vern Paxson, U C Berkeley Icsi, and Mark Allman, *On Dominant Characteristics of Residential Broadband Internet Traffic*, 9th ACM SIGCOMM conference on Internet measurement conference (2009), 90–102.

[MRF+13] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker, *Composing software-defined networks*, Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (Berkeley, CA, USA), nsdi'13, USENIX Association, 2013, pp. 1–14.

[MSB+15] Rodrigo B Mansilha, Lorenzo Saino, Marinho P Barcellos, Massimo Gallo, Emilio Leonardi, Diego Perino, and Dario Rossi, *Hierarchical content stores in high-speed icn routers: Emulation and prototype implementation*, Proceedings of the 2nd International Conference on Information-Centric Networking, ACM, 2015, pp. 59–68.

[NFL+14] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, Peter Steenkiste, and Politecnico Torino, *The Cost of the " S " in HTTPS*, CoNext, 2014.

[nG] nvidia. GTX 580, `http://geforce.com/hardware/desktop-gpus/geforce-gtx-580/`.

[NMN+14] Bruno Astuto A Nunes, Marc Mendonca, Xuan Nam Nguyen, Katia Obraczka, and Thierry Turletti, *A survey of software-defined networking: Past, present, and future of programmable networks*, IEEE Communications Surveys and Tutorials **16** (2014), no. 3, 1617–1634.

[NSS10] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun, *The Akamai network: a platform for high-performance internet applications*, SIGOPS Operating Systems Review **44** (2010), no. 3, 2–19.

[Ope12] Open Networking Foundation, *Software-Defined Networking: The New Norm for Networks [white paper]*, ONF White Paper (2012), 1–12.

[Pat11] Mihai Patrascu, *Unifying the landscape of cell-probe lower bounds*, SIAM J. Comput. **40** (2011), no. 3, 827–847.

[Pav13] George Pavlou, *Information-Centric Networking and In-Network Cache Management: Overview, Trends and Challenges*, 2013, Keynote speech of the 9th IFIP/IEEE Conference on Network and Service Management.

[Pax97] Vern Edward Paxson, *Measurements and analysis of end-to-end internet dynamics*, Ph.D. thesis, University of California, Berkeley, 1997.

[Per98] Charles E. Perkins, *Mobile networking through mobile IP*, IEEE Internet Computing **2** (1998), no. 1, 58–69.

[PGB+14] Diego Perino, Massimo Gallo, Roger Boislaigue, Leonardo Linguaglossa, Matteo Varvello, Giovanna Carofiglio, Luca Muscariello, and Zied Ben Houidi, *A High Speed Information-Centric Network in a Mobile Backhaul Setting*, ICN '14, 2014, pp. 199–200.

[PKV+13] Peter Perešíni, Maciej Kuzniar, Nedeljko Vasić, Marco Canini, and Dejan Kostiū, *OF.CPP: Consistent Packet Processing for Openflow*, Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (New York, NY, USA), HotSDN '13, ACM, 2013, pp. 97–102.

[Plu82] David Plummer, *Ethernet address resolution protocol: Or converting network protocol addresses to 48. bit ethernet address for transmission on ethernet hardware*, RFC 826, 1982.

[Pos81] Jon Postel, *Internet control message protocol*, RFC 792, 1981.

[Pos94] J. Postel, *Domain name system structure and delegation*, RFC 1591, 1994.

[PPK+15] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, Awake Networks, Martín Casado, and Implementation Nsdi, *The Design and Implementation of OpenvSwitch*, NSDI, 2015.

[PV11] Diego Perino and Matteo Varvello, *A Reality Check for Content Centric Networking*, ICN '11, 2011, pp. 44–49.

[PVL14a] D. Perino, M. Varvello, and L. Linguaglossa, *Method And Apparatus To Forward Request For Content*, EP 2947839, ref. 14169327.5, location EU, US, 03 2014.

[PVL+14b] Diego Perino, Matteo Varvello, Leonardo Linguaglossa, Rafael Laufer, and Roger Boislaigue, *Caesar: a content router for high-speed forwarding on content names*, Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems, ACM, 2014, pp. 137–148.

[rg16] The NDN research group, *NDN frequently asked questions*, `http://named-data.net/project/faq/`, 2016.

[Riv76] Ronald L. Rivest, *Partial-match retrieval algorithms*, SIAM J. Comput. **5** (1976), no. 1, 19–50.

[RL95] Yakov Rekhter and Tony Li, *A border gateway protocol 4 (bgp-4)*.

[Rob00] L. G. Roberts, *Beyond Moore's law: Internet growth trends*, Computer **33** (2000), no. 1, 117–119.

[Rou] Route Views Project, *BGP traces*, `http://routeviews.org/`.

[RR06] Eric C Rosen and Yakov Rekhter, *Bgp/mpls ip virtual private networks (vpns)*, RFC 4364, 2006.

[RRGL14] G Rossini, D Rossi, M Garetto, and E Leonardi, *Multi-Terabyte and Multi-Gbps Information Centric Routers*, INFOCOM 2014, 2014, pp. 1–9.

[SAP99] W Richard Stevens, Mark Allman, and Vern Paxson, *Tcp congestion control*, RFC 2581, 1999.

[SDQR10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz, *Runtime measurements in the cloud: Observing, analyzing, and reducing variance*, Proc. VLDB Endow. **3** (2010), no. 1-2, 460–471.

[SH06] Osama Saleh and Mohamed Hefeeda, *Modeling and caching of peer-to-peer traffic*, Network Protocols, 2006. ICNP'06. Proceedings of the 2006 14th IEEE International Conference on, IEEE, 2006, pp. 249–258.

[Sha01] Claude Elwood Shannon, *A mathematical theory of communication*, ACM SIGMOBILE Mobile Computing and Communications Review **5** (2001), no. 1, 3–55.

[SHKL09] Haoyu Song, Fang Hao, Murali S. Kodialam, and T. V. Lakshman, *Ipv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards*, INFOCOM'09 (Rio de Janeiro, Brazil), 2009.

[SNO13] Won So, Ashok Narayanan, and David Oran, *Named data networking on a router: Fast and DoS-resistant forwarding with hash tables*, Architectures for Networking and Communications Systems (2013), no. 1, 215–225.

[SNOS13] Won So, Ashok Narayanan, David Oran, and Mark Stapp, *Named data networking on a router: forwarding at 20gbps and beyond*, ACM SIGCOMM Computer Communication Review, vol. 43, ACM, 2013, pp. 495–496.

[Tan96] Andrew S Tanenbaum, *Computer Networks*, vol. 52, 1996.

[TD99] Mahesh V Tripunitara and Partha Dutta, *A middleware approach to asynchronous and backward compatible detection and prevention of arp cache poisoning*, Computer Security Applications Conference, 1999.(ACSAC'99) Proceedings. 15th Annual, IEEE, 1999, pp. 303–309.

[TFF+13] Patricia Thaler, Norman Finn, Don Fedyk, Glenn Parsons, and Eric Gray, *Ieee 802.1 q*.

[TIAC04] Tomonori Takeda, Ichiro Inoue, Raymond Aubin, and Marco Carugi, *Layer 1 virtual private networks: service concepts, architecture requirements, and related advances in standardization*, IEEE Communications Magazine **42** (2004), no. 6, 132–138.

[VLZL14] Matteo Varvello, Rafael Laufer, Feixiong Zhang, and T.V. Lakshman, *Multi-layer packet classification with graphics processing units*, CoNEXT, 2014.

[VP12] Matteo Varvello and Diego Perino, *Caesar : a Content Router for High Speed Forwarding*, ICN '12, no. Section 5, 2012, pp. 73–78.

[VPL13] Matteo Varvello, Diego Perino, and Leonardo Linguaglossa, *On the Design and Implementation of a wire-speed Pending Interest Table*, NOMEN workshop @ INFOCOM, 2013.

[WHD+12] Yi Wang, Keqiang He, Huichen Dai, Wei Meng, Junchen Jiang, Bin Liu, and Yan Chen, *Scalable Name Lookup in NDN Using Effective Name Component Encoding*, 2012 IEEE 32nd International Conference on Distributed Computing Systems (2012), 688–697.

[WHY+12] Lan Wang, AKMM Hoque, Cheng Yi, Adam Alyyan, and Beichuan Zhang, *Ospfn: An ospf based routing protocol for named data networking*, University of Memphis and University of Arizona, Tech. Rep (2012).

[WLV+12] Yaogong Wang, Kyunghan Lee, Balakrishna Venkataraman, Ravi L Shamanna, Injong Rhee, and Sunhee Yang, *Advertising cached contents in the control plane: Necessity and feasibility*, Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on, IEEE, 2012, pp. 286–291.

[WPM+13] Yi Wang, Tian Pan, Zhian Mi, Huichen Dai, Xiaoyu Guo, Ting Zhang, Bin Liu, and Qunfeng Dong, *Namefilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters*, INFOCOM, 2013 Proceedings IEEE, IEEE, 2013, pp. 95–99.

[WRN+13] Yaogong Wang, Natalya Rozhnova, Ashok Narayanan, David Oran, and Injong Rhee, *An improved hop-by-hop interest shaper for congestion control in named data networking*, SIGCOMM Comput. Commun. Rev. **43** (2013), no. 4, 55–60.

[WZZ+13] Yi Wang, Yuan Zu, Ting Zhang, Kunyang Peng, Qunfeng Dong, Bin Liu, Wei Meng, Huichen Dai, Xin Tian, Zhonghu Xu, Hao Wu, and Di Yang, *Wire speed name lookup: a GPU-based approach*, 2013.

[YC15] Haowei Yuan and Patrick Crowley, *Reliably scalable name prefix lookup*, Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems, IEEE Computer Society, 2015, pp. 111–121.

[YCC14] Haowei Yuan, Patrick Crowley, and Best Case, *Scalable Pending Interest Table Design : From Principles to Practice*, INFOCOM, 2014, pp. 2049–2057.

[YDAG04] L. Yang, R. Dantu, T. Anderson, and R. Gopal, *Forwarding and Control Element Separation (ForCES) Framework*, RFC 3746, 2004.

[YMS13] Wei You, Bertrand Mathieu, and Gwendal Simon, *How to make content-centric networks interwork with cdn networks*, Network of the Future (NOF), 2013 Fourth International Conference on the, IEEE, 2013, pp. 1–5.

[YMT+12] Wei You, Bertrand Mathieu, Patrick Truong, Jean-Francois Peltier, and Gwendal Simon, *DiPIT: A Distributed Bloom-Filter Based PIT Table for CCN Nodes*, 2012 21st International Conference on Computer Communications and Networks (ICCCN) (2012), 1–7.

[YSC12] Haowei Yuan, Tian Song, and Patrick Crowley, *Scalable NDN Forwarding: Concepts, Issues and Principles*, 2012 21st International Conference on Computer Communications and Networks (ICCCN) (2012), 1–9.

[YTG13] Soheil Hassas Yeganeh, Amin Tootoonchian, and Yashar Ganjali, *On scalability of software-defined networking*, Communications magazine, IEEE **51** (2013), no. 2, 136–141.

[ZEB+10] Lixia Zhang, Deborah Estrin, Jeffrey Burke, Van Jacobson, James D Thornton, Diana K Smetters, Beichuan Zhang, Gene Tsudik, Dan Massey, Christos Papadopoulos, Lan Wang, Patrick Crowley, and Edmund Yeh, *Named Data Networking (NDN) Project*, Tech. Report October, 2010.

[Zim80] Hubert Zimmermann, *OSI reference model - the ISO model of architecture for open systems interconnection*, Communications, IEEE Transactions on **28** (1980), no. 4, 425–432.

[ZLL13] Guoqiang Zhang, Yang Li, and Tao Lin, *Caching in information centric networking: a survey*, Computer Networks **57** (2013), no. 16, 3128–3141.

[ZZY+14] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat, *Libra: Divide and conquer to verify forwarding tables in huge networks*, Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (Berkeley, CA, USA), NSDI'14, USENIX Association, 2014, pp. 87–99.