TagNet: A Scalable Tag-Based Information-Centric Network

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Michele Papalini

under the supervision of
Antonio Carzaniga

October 2015

Dissertation Committee

Dissertation accepted on 23 October 2015

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Michele Papalini
Lugano, 23 October 2015

To my family

Abstract

The Internet has changed dramatically since the time it was created. What was originally a system to connect relatively few remote users to mainframe computers, has now become a global network of billions of diverse devices, serving a large user population, more and more characterized by wireless communication, user mobility, and large-scale, content-rich, multi-user applications that are stretching the basic end-to-end, point-to-point design of TCP/IP.

In recent years, researchers have introduced the concept of Information Centric Networking (ICN). The ambition of ICN is to redesign the Internet with a new service model more suitable to today's applications and users. The main idea of ICN is to address information rather than hosts. This means that a user could access information directly, at the network level, without having to first find out which host to contact to obtain that information.

The ICN architectures proposed so far are based on a "pull" communication service. This is because today's Internet carries primarily video traffic that is easy to serve through pull communication primitives. Another common design choice in ICN is to *name* content, typically with hierarchical names similar to file names or URLs. This choice is once again rooted in the use of URLs to access Web content. However, names offer only a limited expressiveness and may or may not aggregate well at a global scale.

In this thesis we present a new ICN architecture called TagNet. TagNet intends to offer a richer communication model and a new addressing scheme that is at the same time more expressive than hierarchical names from the viewpoint of applications, and more effective from the viewpoint of the network for the purpose of routing and forwarding.

For the service model, TagNet extends the mainstream "pull" ICN with an efficient "push" network-level primitive. Such push service is important for many applications such as social media, news feeds, and Internet of Things. Push communication could be implemented on top of a pull primitive, but all such implementations would suffer for high traffic overhead and/or poor performance.

As for the addressing scheme, TagNet defines and uses different types of addresses for different purposes. Thus TagNet allows applications to describe information by means of sets of tags. Such tag-based descriptors are true content-based addresses, in the sense that they characterize the multi-dimensional nature of information without forcing a partitioning of the information space as is done with hierarchical names. Furthermore, descriptors are completely user-defined, and therefore give more flexibility and expressive power to users and applications, and they also aggregate by subset.

By their nature, descriptors have no relation to the network topology and are not intended to identify content univocally. Therefore, TagNet complements descriptors with locators and identifiers. Locators are network-defined addresses that can be used to forward packets between known nodes (as in the current IP network); content identifiers are unique identifiers for particular blocks of content, and therefore can be used for authentication and caching.

In this thesis we propose a complete protocol stack for TagNet covering the routing scheme, forwarding algorithm, and congestion control at the transport level. We then evaluate the whole protocol stack showing that (1) the use of both push and pull services at the network level reduces network traffic significantly; (2) the tree-based routing scheme we propose scales well, with routing tables that can store billions of descriptors in a few gigabytes thanks to descriptor aggregation; (3) the forwarding engine with specialized matching algorithms for descriptors and locators achieves wire-speed forwarding rates; and (4) the congestion control is able to effectively and fairly allocate all the bandwidth available in the network while minimizing the download time of an object and avoiding congestion.

Contents

Figures

Tables

Chapter 1

Introduction

In this thesis we introduce an new information-centric network (ICN) called *TagNet*. TagNet has two main distinguishing features, as compared with the other mainstream ICN architectures.

First of all, TagNet supports a truly *information* centric addressing. This is not the case in the mainstream ICN architectures based on hierarchical names. In fact, those architectures are built under the assumption that names contain a globally routable prefix that effectively plays the role of an IP address or host name in a URL. In other words, names would still associate the content with a particular host or location. By contrast, TagNet provides a multi-modal addressing scheme with application-defined addresses that can be truly location-independent (but can also represent locations if those are meaningful to the application) and with network-defined addresses that allow for very efficient matching and forwarding operations.

The second main feature of TagNet is that it combines the best of the mainstream "pull" ICN architectures, with the flexibility of a "push" publish/subscribe event notification service at the network level. This allows for very efficient pull flows that can be used, for example, in video streaming, and low-latency push notifications, which are essential in new technologies such as the so-called Internet Of Things.

In essence, this PhD thesis describes "systems" work. We formulate a network architecture that we believe would be useful and significantly more expressive for applications, and we sudy its feasibility by implementing it and by evaluating it experimentally. In particular, we develop an entire stack of protocols for TagNet. We describe a routing protocol based on spanning trees, that allows easy implementation of multicast and anycast forwarding, as well as an efficient routing state compression that is essential for scalability on large networks. We then propose a forwarding engine for TagNet, that, thanks to the multiple addresses provided by TagNet, can achieve line speed throughput using forwarding tables with state generated by hundreds of millions of users. Finally, we propose a transport protocol for pull flows that effectively uses multiple paths to maximize the bandwidth for each flow, avoiding congestions.

This introduction chapter is structured as follows: In Section 1.1 we describe why the research community started to look into the idea of ICN, and how ICN should solve some of the problems that are common in today's IP networks. In Section 1.2 we briefly describe the history and the evolution of ICN architectures and then we focus on the presentation of Content-Centering Networking (CCN), which is one of the most relevant and influential ICN architecture, and also the one we use as a reference for comparison with TagNet. In Section 1.3 we highlight some of the issues and limitations of the mainstream ICN architectures, and finally, in Section 1.4 we describe the contribution of this thesis.

1.1 From Host-Centric to Information-Centric Networking

Networks are designed to be host-centric. The main functionality of the network is to connect two end-points, each identified by an address assigned by the network. Each packet contains the address of the destination host in its header, which the network uses to forward the packet towards the destination. This is the basic principle of packet switching introduced by ARPANET and still at the foundation of today's TCP/IP protocol stack.

When this host-centric communication model was designed, networks were small, and their main application was a remote terminal or perhaps file transfer that would allow users to share computing resources such as large and expensive mainframe computers. In such a context, the host-centric model works very well. Furthermore, security was not an issue, since only a few trustworthy organizations and users were granted physical access to the infrastructure. This context lead to a design that was conceptually simple and, in part due to this simplicity, also extremely effective and successful.

Today the Internet is much bigger than the original ARPANET, and it is accessible by billions of users. This enormous growth introduced serious problems of scalability and security. Also, the way we use the network is different today: the Internet is mostly a content distribution network, not just a mechanism to connect two hosts. In fact, the vast majority of traffic nowadays consists of video distribution. According to Cisco,[1] video traffic was 66% of the total Internet traffic in 2013, and will reach 79% by 2018. A typical example of video distribution application is YouTube, where multiple publishers provide content that can then be retrieved by many users at the same time, and there are similar *live* broadcasting services (e.g., Twitch.tv). This breaks the basic assumption that the communication is point-to-point, simply because the communication process involves multiple users acting as producers and consumers (or both).

Another new challenge comes from the introduction of new technologies together with the cultural changes they induce. The advent of wireless data transmission introduced a new way to connect to and use the Internet whereby users are free to move while remaining connected. Mobile traffic is growing by 57% every year,[2] and this level of user mobility is not well supported by the basic host-centric communication model.

In reaction to these changes, the research community introduced new protocols and techniques to handle new technologies and applications within the existing network infrastructure. To handle the growth of the Internet they developed IPv6, featuring a much expanded address space and a more modular design. Classless inter-domain routing, or CIDR, is another example of a method adopted to reduce the size of routing and forwarding tables. Thanks to this form of aggregation, the size of the routing tables is reduced on average by almost 45% and 25% for IPv4 and IPv6, respectively.[3]

Security has also emerged as a primary concern for the design and use of the Internet, which led to the development of a wide range of protocols and systems, some of which are quite mature and widespread. As an example, consider today's use of HTTPS, which according to Naylor et al. has now reached 50% of all web traffic [57].

New protocols and systems were also developed to handle new technologies and usage patterns such as wireless channels and mobility. Mobile IP is an example: it allows users to keep a connection alive even when they move among different access points.

Another important change that somehow challenges the original network design is the introduction and widespread use of Content Delivery Networks

[1]The Zettabyte Era: Trends and Analysis. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.pdf

[2]Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2014–2019. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.pdf

[3]CIDR Report. http://www.cidr-report.org/as2.0/; IPv6 CIDR Report. http://www.cidr-report.org/v6/as2.0/.

(CDN). CDNs are heavily used today, as they are fundamental in reducing traffic and improving the quality of service for end-users. As an example, consider that Akamai, one of the primary CDNs, handles 20% of the total web traffic with about 30 million hits per seconds.[4] CDNs break the end-to-end principle by introducing essential in-network processing such as transparent proxies, transcoding, and compression.

All the patches to the original network architecture make the architecture and its stack of protocols more complex, and also introduce inefficiencies in traffic management. A very current example is given by the HTTPS protocol. Since HTTP hides all application-level headers and content, it also prevents ISPs from caching, compressing, and optimizing content in-transit. Naylor et al. attempt to measure the cost of encryption [57] and report that ISPs have seen a large increment in upstream traffic due to the fact that they can not serve requests locally using proxies for transparent caching. This generates additional traffic that in turn requires additional network capacity. HTTPS also affect the quality of service perceived by users due to the additional latency of the TLS handshake. Similar problems are common also in other protocols such as Mobile IP, which requires a longer path from the sender to the receiver compared to a direct connection (triangular routing problem [64]), which in turn increases traffic and latency.

It is in this scenario that many researchers have proposed new network architectures, often referred to as Future Internet Architectures, better suited to serve diverse communication and usage patterns for today's and hopefully tomorrow's Internet. One such architecture is that of a network capable of distributing content to interested users, and of reducing traffic by pushing and caching content closer to users. In these networks the content itself, as opposed to the end-nodes, becomes the central focus of the addressing scheme. Thus a user does not specify a host address in his or her requests, simply because most of the time the user does not care about the source of the content. Instead, the user addresses the content directly by specifying a name or a description of that content. The network is then responsible for routing the packet in the right direction to get a copy of the required data, using only the name or description provided by the user as an address. This new vision of the network, where the primary network function is centered on the content, is called Information Centric Networking (ICN).

Again, the main idea behind ICN is to address information itself, instead of

[4]Visualizing Global Internet Performance. https://www.akamai.com/us/en/solutions/intelligent-platform/visualizing-akamai/index.jsp

addressing the hosts involved in the communication process. This simple change of perspective gives us many possible advantages over a traditional host-centric network. One such advantage is with respect to security, or more specifically integrity and authentication. Since information is the central unit of addressing, we can simply secure (sign) the content *once* instead of securing all the end-to-end connections through which the content is transferred between hosts. With such secured content, routers could also store and later serve that same content from in-network caches. Another advantage of ICN is that addresses are location-independent, since they point to a piece of information rather than a device. Therefore, a user could download different pieces of the same content from different sources, reducing download time and increasing the perceived quality of service. The network is also more reliable: if a link breaks or a producer fails to provide the data, the same content could be found somewhere else, without the need to update the forwarding tables. The usage of location-independent addresses is also useful in case of mobile connections, because mobile end-users can be easily supported by the network without any additional protocols or infrastructure such as that of mobile IP.

1.2 Background: ICN History and Architectures

The idea of routing packets on names or according to their content is not new, and in fact it goes back to almost 15 years ago. Carzaniga et al. proposed the notion of a content-based network in 2001 [19], and later developed this notion with specific routing and forwarding schemes [20, 18]. Within this notion, Carzaniga et al. propose to route traffic using predicates, namely sets of attributes and conditions on their values that describe the content of a packet. Also in 2001, the TRIAD project proposed to address packets using names similar to URLs [41].

In the following years, other projects and systems such as i3 [75], VRR [9] and ROFL [10], proposed to use DHT overlay networks to address content. Along the same line, Koponen et al., with the DONA system in 2007, propose to identify content objects with flat names, similar to hash keys and then to use a name resolution network to map a required content name to its location, in a similar way as DNS associates URIs to IP address. The PSIRP/PURSUIT project [77] also proposes a similar architecture, this time using a hierarchical name resolution network to associate content names, identified with flat labels, with content locations. Perhaps the most interesting contribution of this project is a new forwarding scheme called LIPSIN that replaces the traditional IP forwarding in local networks [48].

The project that perhaps more than others brought renewed attention to the idea of information centric networking is the Content Centric Networking (CCN) project originated at PARC around 2006 [47]. This project also evolved into another mainstream ICN project called Named Data Networking (NDN). The CCN/NDN architecture addresses content using hierarchical names similar to URLs (or file names in a Unix-like file system). Since this architecture is so prominent, and also because it is very relevant for this thesis, we describe it extensively in the next section. After the initial CCN proposals, several other projects, including NetIfn/SAIL [2] and MobilityFirst [67], researched the the notion of information centric networking. Ahlgren et al. and Xylomenos et al. have surveyed many of these projects and ideas [1, 85].

1.2.1 Content-Centric Networking Architecture

In this thesis we refer many times to CCN/NDN, which we consider the most mature ICN architecture. The first project originated at PARC, and nowadays important networking companies like Cisco, Alcatel, and Orange, in addition to different universities, are working on this architecture. Since this thesis is related to CCN/NDN, we present a brief overview of the CCN/NDN architecture.

Jacobson et al. introduced the first proposal of Content-Centric Networking (CCN) [47]. After the end of the first CCN research program, a new one was founded under the name of Named-Data Networking (NDN), while CCN was still active at PARC. Recently, the CCN and NDN projects have adopted different approaches regarding some architectural components, as well as many implementation details. Still, the basic architectural principles remain common to both projects and are an important basis for the work described in this thesis. Since this common basis originated in the CCN project, we will refer to it as CCN, and in cases where there are significant differences between the two architectures, we will specify the right architecture in the text.

In CCN, the addressing scheme is based on names. That is, CCN transmits named content. In particular, CCN uses hierarchical names similar to URLs. Each piece of content is divided into chunks, which are the basic addressable unit. Therefore, each chuck has its own name. For example the name *ch/usi/-papalini/thesis.pdf/v2/c4* might refer to a chunk of this thesis, in particular to the fourth chunk (c4) of version 2 of the thesis (v2). Names can be aggregated in forwarding tables according to their hierarchical structure. This means that a set of names can be represented by their common prefix in a similar way as IP prefixes.

CCN is a pull-based architecture: a user who wants to get some content must explicitly request that content. Thus there are two kinds of packets: *interest* packets carry a request for a specific content chunk, and *data* packets carry the requested content chunk back to the user.

Forwarding Information Base (FIB)

Prefix	Output Ifx
/ch/usi/*	6
/com/youtube/*	1,7
...	...
...	...

Pending Interest Table (PIT)

Name	Input Ifx
/ch/usi/thesis.pdf/v2/c3	2
/ch/usi/thesis.pdf/v2/c4	2
/com/youtube/video.mp4/v3/c2	4,10
...	...

Content Store (CS)

Name	Data
/ch/usi/thesis.pdf/v2/c1	...
/ch/usi/thesis.pdf/v2/c2	...
/com/youtube/video.mp4/v3/c1	...
...	...

Figure 1.1. State on a CCN router

Every router in the network has three tables: a *Forwarding Information Base (FIB)*, a *Pending Interest Table (PIT)* and a *Content Store (CS)*, which is generally called cache. Figure 1.1 shows the three tables.

The Forwarding Information Base (FIB) is the table used to forward interest packets toward one of the sources of the content requested by that interest. In particular, the FIB associates a prefix with a set of output interfaces through which the router may forward an interest packet matching that prefix. Similar to the FIB in an IP router, forwarding works as a longest-prefix matching (LPM) between the name in the interest packet and the prefixes in the FIB. Since there may be multiple sources for the same content, a prefix may be associated with multiple output interfaces, as is the case for the */com/youtube/** prefix in Figure 1.1. In this case, the router may choose one or more of the matching interfaces.

The Pending Interest Table (PIT) stores temporary state for interests that were forwarded by the router but not yet satisfied by a corresponding data packet. An interest leaves some state in the PIT of each router along its forwarding path. This state is then used to route the corresponding data packet back to the requesting user. For each interest packet, the router must store in the PIT the entire name of the interest plus the interface through which the interest reached the router (denoted *input ifx* in Figure 1.1). This way, a data packet always follows the reverse path of the corresponding interest. For example, the PIT of Figure 1.1 indicates that the router received an interest packet for */ch/usi/papalini/thesis.pdf/v2/c3* from interface 2. Therefore, the router will forward the corresponding data packet through interface 2. If the router receives an interest with a name that is already in the PIT, the router can aggregate the two interests. In particular, the router does not forward the second interest packet, but adds the incoming interface of the new interest to the existing entry in the PIT. In the example of Figure 1.1, this is what happened in the case of the interest */com/youtube/video51.mp4/v3/c12*.

The Content Store (CS), or cache, may store a copy of the data packets received by the router. The table is indexed by name. If the router has some content that satisfies an interest in its cache, the router replies directly to that interest on behalf of the content provider with a data packet containing the cached copy. The router may apply different replacement policies to manage its cache.

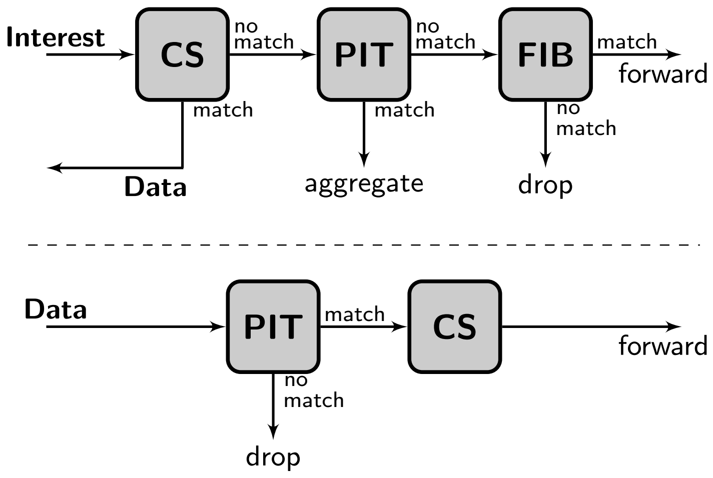

Figure 1.2. Forwarding phases on a CCN router: interest packet forwarding (top) and data packet forwarding (bottom)

A CCN router forwards interest and data packets in two different ways, as illustrated in Figure 1.2. An interest is forwarded in at most three phases (top part of Figure 1.2). In the first phase the routers checks if the required content is in the content store. If so, the router replies immediately to the user, and no further processing is needed. If the content store does not have the content, the router proceeds with the second phase. In this phase, the router checks for a PIT entry with the same name of the interest packet. If the entry exists, the router aggregates the interest to the existing entry and does not forward the interest. Otherwise, the router proceeds with the third and last phase. Here the router updates the PIT, introducing a new entry for the interest, and forwards the packet according to the longest-prefix matching in the FIB.

The forwarding procedure for data packet requires two steps, as illustrated in the lower part of Figure 1.2. First, the router looks for a PIT entry related to the received data packet. If no such entry exists, then the router drops the packet, because it does not know where to forward the packet. Otherwise, if a corresponding PIT entry exists, the router stores the content of the packet in the content store (possibly), then removes the entry from the PIT, and proceeds to forward the packet to all the interfaces listed in the PIT entry.

1.3 ICN Issues and Limitations

Although the ICN idea attracted a lot of attention and there is a lot of research on this topic, many problems are still open and they remain without a clear answer. Here we highlight some of the problems that are relevant for this thesis.

One of the most critical decisions in the design of an ICN architecture is the choice of an appropriate naming scheme. Much of what happens in a router, throughout the whole network, and also at the application level, depends crucially on this choice. For example, naming affects in a fundamental way forwarding and matching, as well as routing and route aggregation, and therefore the scalability of the whole architecture. But naming also determines the expressive power of the service offered to applications and users, and there is a clear tension between, on the one hand the expressiveness of the service at the application level, and on the other hand its scalability at the network level.

In the literature, there are basically two main proposals for naming: *flat names* and *hierarchical names*. Flat names are labels with really basic structure, or rather no structure at all. Flat names are usually not intended to be readable by humans, and can be seen as random identifiers. As an example, in DONA [51] a name has two parts, called P and L. P is the cryptographic hash of the principal public key, where the principal for a content is the node that first published such content. L is a label that uniquely identifies the content within the principal. A similar kind of names is used in PSIRP/PURSUIT [77]. The main benefit of these "flat" names is that they are easy to match, simply because matching means *exact* match, which admits to very efficient algorithms. Such flat labels are also considered “self-certified” names [27], meaning that they can be built and serve as cryptographic signatures for the content. In this way, when a user receives a packet, the user can also check if the content is authentic. However, this notion of self-certified names begs the question of how an application would be able to issue a request for presumably unknown content.

Hierarchical names are more structured and, as seen in the examples of Section 1.2.1, they are similar to URLs. They are intended to be understandable by humans, and they can be easily used to address some content by an application. However, these names require a more complex algorithm to find the longest matching prefix from a possibly large forwarding table. This has been one of the major concerns with CCN, but as demonstrated by several authors [82, 73, 63, 81], it is possible to perform longest-prefix matching on names at wire speed.

The CCN architecture [47] also includes important authentication features. Contrary to flat names in DONA, hierarchical names in CCN do not themselves carry authentication information. Instead, this information is attached to the data packet as a signature. This security feature is based on standard public-key signatures that allow consumers as well as routers to verify the integrity and provenance of content. Although this security feature is commonly accepted, it still poses problems in distributing and authenticating public keys [54]. It also raises the question of performing public-key signature validation in routers, although once again this is not a serious conceptual obstacle.

One of the main criticisms of ICN is related to its scalability. In fact, at this point there is little or no evidence that any of the proposed architectures could scale to global-size networks. According to Ghodsi et al., a global ICN must be able to handle at least 10^{12} objects [38], which is a prohibitive size for forwarding tables for current and perspective technology. It is therefore essential to compress forwarding information through some form of aggregation.

All architectures based on flat names deal with this scalability problem by using look-up functions that are external to the forwarding process, for example through specialized data structures such as DHTs. In other words, those architectures externalize the mapping between names, which are user-defined quantities, and an underlying network address (e.g., IP) that is then used for forwarding. Some of those architectures also rely on name-scoping techniques to further increase scalability [77].

Hierarchical names in CCN/NDN instead are directly used for forwarding, and therefore pose a serious question of aggregation. The aggregation of CCN names is analogous to prefix aggregation used for IP addresses. However, it is still unclear whether that aggregation would be effective, since contrary to

IP addresses, names are user-defined and may not have any relation with the network topology.

Another problem that involves the design of the naming scheme is the expressiveness of the names. In all proposed naming schemes, a name can refer to only one piece of content. This unique match, one name one content, is useful in some applications, like a file transfer, where a users wants to be specific and address one specific file. However, there are other applications, that are commonly used, that needs a more sophisticated way to address content. An example is a news feed, where a user wants to get all the news related to a particular topic that may come from multiple sources. Unfortunately, there is no easy way to express such an interest using the kind of addresses existing in the literature with a single name.

In CCN, all the information related to the packet are embedded in the name. This can be really inefficient in some cases, because each function that processes the packets needs to parse the entire name in order to find the relevant set of components. In addition, PITs need to store the entire name of each interest, so, if the name is too long, the size of these tables would grow really quickly. Moiseenko et al. show that a name design where all the information is embedded in the name may be problematic [55]. In particular, Moiseenko et al. study different ways to implement the HTTP protocol in NDN, showing that it is not a trivial thing to do. To solve this problem, they propose to introduce additional files that are specific to some application or protocol in order to avoid to put all the data in the name. This problem is less severe in CCN after the introduction of labeled content information,[5] but is still present in NDN.

The last problem that we want to highlight is related to the communication model. All the proposed ICN architectures support only *pull* communication, meaning that a user needs to request each piece of information the user wants to receive. This preference for pull communication is understandable, since most of the traffic in the Internet can be retrieved on-demand by users. In particular, this is true for video content, which makes up the majority of Internet traffic today. However, many applications, such as social networks or news feeds, need to spread many short messages to the interested users in a short time, because many of the messages are valuable only for a short period. And the best way to send this kind of messages is to provide a *push* primitive, where the producer directly contact the interested user. Unfortunately, push communication is not easy to implement using the ICN designs proposed so far.

[5]Labeled Content Information. http://www.ietf.org/id/draft-mosko-icnrg-ccnxlabeledcontent-00.txt

1.4 Contribution and Structure of the Thesis

In this thesis we tackle the problems described in the previous section by introducing a new ICN architecture called *TagNet*. We design our ICN with three architectural goals in mind:

1. TagNet should provides rich communication primitives natively.
2. The naming scheme has to be flexible. It has to allow users to express their interests, but, at the same time, it needs to be optimized for each packet processing function at different levels in the network.
3. The architecture should be able to scale at a global level.

TagNet provides natively rich communication primitives, meaning it allows both push and pull communication. We show that this can be done easily, without introducing additional state in routers and without changing the forwarding procedures. Specifically, we show how to forward packets using a single FIB in both push and pull mode, avoiding loops, and allowing various forms of multicast as required by the application.

Our naming scheme is more complex and structured as compared with all other ICN proposals. We decided to have different kinds of addresses with different purposes that can all coexist in the same packet. We see this apparent added complexity as a principled architectural decision and also a fundamental architectural improvement. In particular, users and applications can use *descriptors* to describe what they request from the network. We implement descriptors as sets of tags, although the architecture could accommodate other forms of descriptors. Tag sets allow users to target a single file, like in the case of hierarchical names, or to request a broader set of information available in the network. Besides descriptors, in TagNet we also explicitly define another kind of address that we call *content identifier*. As the name suggests, this kind of address is bound one-to-one with a unique content chuck. Content identifiers play a fundamental role for transport protocols and also for in-network caching. The third and last form of address we use in TagNet is what we call *network locators*. These are network-defined addresses intended exclusively for host-based forwarding.

The last and perhaps more ambitious goal of our architecture is to achieve Internet-level scalability. This is possible thanks to the aggregation of tag-based descriptors, as well as the specialization of addresses, in particular locators. In particular, regarding aggregation, notice that tag sets aggregate by subset, which is a more powerful aggregation function than prefix aggregation provided by

CCN/NDN. Our evaluation shows that we can fit more than 10 billion names in a few gigabytes of space and we can also handle updates efficiently.

The structure of the thesis is as follows: in Chapter 2 we present the architecture: we introduce the communication primitives provided by TagNet, as well as its naming scheme. In Chapter 3 we describe the routing protocol that we developed for TagNet. Chapter 4 describes the matching function and the forwarding engine developed for TagNet. Chapter 5 describes a transport and congestion-control protocol that is mostly based on CCN but still relevant for TagNet. Finally, in Chapter 6 we draw some conclusions and speculate on future work.

Chapter 2

TagNet: a New ICN Architecture

In this chapter we present TagNet, our ICN architecture. This chapter describes the architectural design of our network. First of all, we describe our service model that supports natively both push and pull communication flows. We also argue *why* it is important to have such a rich set of communication primitives, and in support of that argument we present a series of applications that can benefit from those primitives. Furthermore we show that adding support for push flows does not require additional state within routers.

In this chapter we also describe the naming scheme that we adopt for TagNet. We argue in favor of a rich and expressive way to address content, but we also argue that such expressive form of addresses should not be abused, and instead that it is important to have other, specialized addresses for different purposes.

We support these arguments with a comparative evaluation of our architecture with the original proposal of CCN. This evaluation shows that a network that supports richer communication primitives generates less traffic and works properly under various types of workloads.

This chapter is structured as follow: in Section 2.1 we argue why a rich communication service is important and how we can support it in TagNet; in Section 2.2 we present our "naming" scheme consisting in fact of multiple and quite diverse forms of addresses; in Section 2.3 we evaluate TagNet against CCN.

2.1 Communication Primitives: Push vs Pull

All the proposed ICN architectures allow only *pull* communication: a user receives some content only when he or she requests such content. We call this user-initiated communication *on-demand content delivery*. As we already discussed in the introduction, the main reason behind this architectural choice is that most of the Internet traffic is generated by applications, such as video streaming, that work very well with this model. On-demand content delivery is the best way to access *persistent* information, meaning information that is valuable for a long period of time, because it does not change. Video, music, and pictures are examples of persistent information.

However, there is an other set of applications where users want to receives specific information, related to a particular topic or published by a particular user, in more or less real time. A user expects to get new content without asking for updates every time. Applications with these characteristics are micro-blogs (e.g., Twitter), social networks, and news feeds. We discuss more about these applications in the next section. However, we observe that, generally speaking, these applications exchange *ephemeral* content, meaning content that is valuable for a short period of time. The best way to access this ephemeral information is to decouple the consumers from the producers, using a publish/subscribe communication pattern [19, 28]. This pattern is based on *push* communication, where the producers start the communication flow, sending the information to all the interested users as soon as it is available. We call this communication model *publish/subscribe event notification*.

In today's internet, the amount of traffic generated with publish/subscribe event notification is much lower than the traffic generated by on-demand applications. This is due in part to the fact that "events" themselves usually carry a small amount of information (possibly with pointers to larger objects), and also in part to the fact that publish/subscribe event notification is not natively supported by most of the Web infrastructure. And this is why most of the ICN architectures do not support a push communication primitive. However, applications such as social networks and microblogging are becoming more and more popular, and, for this reason, the number of messages exchanged with publish/-subscribe mechanisms could become quite high. High frequency event notifications can generate a lot of overhead, both in terms of traffic and data processing, if they are not not handled correctly. We think that an event notification service should be supported natively from an ICN architecture.

2.1.1 Publish/Subscribe Event Notification: Applications

We now analyze some possible use-case scenarios for publish/subscribe event notification in toady's Internet. Other applications that are suited for this communication primitive are described by Carzaniga et al. [17].

The first application we want to discuss is the Web itself. There is in fact a lot of work done by the Web programmers community in order to deploy a "push" version of the Web. The idea is to allow the server side to push data to the users as soon as the data are available, avoiding polling over HTTP. This has the advantage of reducing the load on the server, as well as the data latency. A collection of these techniques is known as Comet.[1]

BOSH (Bidirectional streams Over Synchronous HTTP) is a protocol that allows push communication and is used as support to the XMPP protocol, which is an important standard for presence and messaging. BOSH implements a long polling whereby the client sends an HTTP request to the server, and the server then replies only when it has data to send. Every time new data is delivered, the server closes the connection, so that the client needs to issue a new request.

Another technology that allows a server to push content to the client is Server-Sent Event (SSE). SSE uses HTTP to establish a persistent connection between the client and the server. The server is then able to push updates, avoiding continuous polling from the client, reusing every time the same connection. This protocol is part of the HTML5 standard and is implemented by most popular browsers.

WebSocket is a more recent technology that allows push from the server as well as data transfer from the client. A WebSocket provides a full-duplex communication channel between the server and the client, essentially by reverting an HTTP connection back to its TCP base: the client connects using HTTP, issuing a special HTTP request that instructs the server to treat that connection as a basic TCP connection. This connection can then be used by the server to push data to the client at any time. Like SSE, the WebSocket protocol is part of the HTML5 standard, and it is supported by many browsers and servers.

All these technologies are used in the so called real-time web, that forwards content to the users as soon as it is published. There are many frameworks that can be used to develop applications with these features. Some examples are Fanout,[2] Hydna,[3] PubNub,[4] Pusher,[5] and others.

Going back to a more concrete application, one of the most common use-case scenario for a publish/subscribe event notification are social networks. Facebook has more that 1 billion active users monthly,[6] while Twitter is much smaller, but generates 500 million tweets per day.[7] In a social network users want to get in-

[1]Comet Daily. http://cometdaily.com/
[2]Fanout webpage. https://fanout.io/
[3]Hydna webpage. https://www.hydna.com/
[4]PubNub webpage. http://www.pubnub.com/
[5]Pusher webpage. https://pusher.com/
[6]Facebook Statistics. http://newsroom.fb.com/company-info/
[7]Twitter Statistics. https://about.twitter.com/company

formation according to their interests, or from their friends. A publish/subscribe system is perfectly suited for these applications because a user does not need to know from where and when he will receive the next message.

Push communication can also be useful on devices where energy consumption is an issue, such has mobile devices. A device that listens on a single port, waiting for messages pushed by a server, consumes much less energy than a device that needs to constantly poll different servers to get updates. This is the idea behind Google Cloud Messaging on Android devices,[8] and Apple Push Notification Service on iOS and OS X devices.[9] Both these services are based on XMPP. Using these services, users needs to maintain only one listening socket through which they can receive notifications immediately and also efficiently, reducing energy consumption as well as traffic.

Many applications for the so-called Internet of Thing (IoT) can also benefit from a push-based communication model. In this scenario, the network consists of devices many of which are very much constrained in processing and energy budget. A communication protocol such as publish/subscribe that can reduces traffic and therefore energy consumption is an important feature to support IoT. In fact, nowadays IoT networks already use publish/subscribe protocols, such as the Message Queue Telemetry Transport (MQTT).[10] There are many proposals to use ICN protocols, and in particular CCN, in this networks, but the lack of a push primitive is generally a problem. As a result, some researchers propose to relax the CCN architectural constraints in order to define a way to better handle push traffic transmission [33, 4].

It is important to notice that some of the applications described here are still implemented in a pull-based fashion, even though, as we saw, there are many frameworks to enable pull-based applications. That is due to the fact that HTTP, which is the most common protocol to transfer data in the Internet, implements a pull communication flow. However, push communication is getting more e more important for today's Internet usage, and since ICN provides us a way to rethink and redesign the entire Internet architecture, we should use this occasion to rebuild the protocol stack in such a way as to better fit future users needs.

Other researchers in the ICN community share this view, which is expressed in part in the ICNRG draft "Enabling Publish/Subscribe in ICN".[11]

[8]Google Cloud Messaging. http://developer.android.com/google/gcm/gcm.html

[9]Apple Push Notification Service. https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Chapters/ApplePushService.html

[10]MQTT webpage. http://mqtt.org/

[11]Enabling Publish/Subscribe in ICN. http://tools.ietf.org/pdf/draft-jiachen-icn-pubsub-01.pdf

2.1.2 Do We Really Need a Push Primitive?

In the previous section we argued that an ICN architecture should provide both on-demand content delivery and publish/subscribe event notification natively at the network level. This way, all kinds of communication flows can be easily handled by the network, thereby reducing unnecessary overhead traffic.

More specifically, although we claim that it is important to implement push communication, we have not yet demonstrated that such a communication pattern has to be a network *primitive*. In fact, it is possible to emulate push primitives using only pull communication. In this section we want to argue that implementing push communication on top of pull communication primitives is not a good design strategy, due to the semantic mismatch between the two forms of communication. Generally speaking, implementing publish/subscribe event notification using only on-demand content delivery primitives leads to high control traffic and, under certain workloads, unexpected and potentially erroneous application behaviors.

Considering CCN, there are at least two ways to implement publish/subscribe event notification without modifying the original architecture. The first implementation uses *polling*. The second, that we call *interest as notification*, uses interests packets to notify interested consumers that some new information is available.

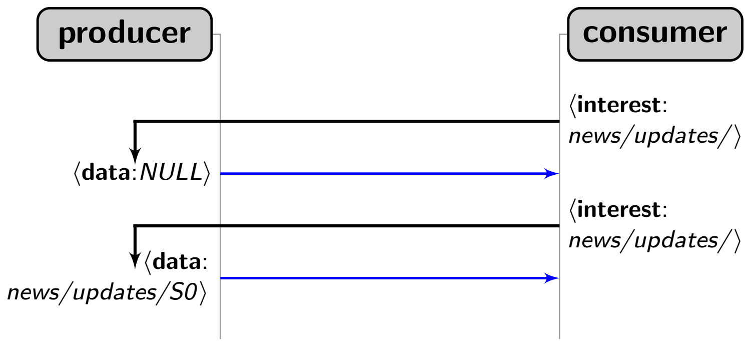

Figure 2.1. Publish/subscribe event notification implemented with polling

In the polling implementation, illustrated in Figure 2.1, a consumer issues interests at a more or less constant rate to check if new content is available at the producer. If the producer does not have any update, the producer replies with an empty message, like in the case of the first interest in the picture. In the other case, when the producer has some events to notify, the producer replies with the corresponding event data, like in the case of the second interest of the picture.

Polling is efficient only in the case where the polling rate is similar to the publication rate. In this case we can get events in time, and without much traffic overhead. However this scenario is rare, mainly because events are asynchronous and may be quite irregular. Therefore, if the polling rate is too low, it is possible to miss some notifications. Or, on the other hand, if the polling rate is to high, the application would generate a lot of useless traffic.

In CCN we also need to take into account the PIT expiration time associated with each interest. A polling rate higher than the expiration time (or rather its inverse) is useless, because repeated interests with the same name would be aggregated in the PIT (see Section 1.2.1 for PIT aggregation). Selecting a polling rate that is close to the PIT timeout (inverse) can be a good decision. However, this can lead to scalability problems for the PITs. In fact, issuing an interest as soon as the same previous interest expires is similar to creating a persistent entry in the PITs, which are not conceived for, nor designed to handle such a situation. Also, the PITs aggregate interests only by exact name match, so in other words, persistent interests do not aggregate well, which once again means that the PITs may suffer from scalability problems. And as shown by Virgilio et al., when the PITs are overloaded the performance of the network degrade, and, depending on the PIT implementation, the network may be unable to deliver packets [79].

Another problem with the polling implementation is when there are multiple producers for the same kind of events and the same name prefix. In this case a user has to contact all the producers independently, because the producers may generate different events at different times. In this scenario, the polling implementation induces yet more traffic, as well as more state in the PITs.

An even more serious issue comes from the forwarding strategy of the routers in CCN. Usually an interest is forwarded only to one interface. A router may implement a “forwarding strategy,” meaning an algorithm that ranks different interfaces for the same prefix according to different parameters (e.g., response time), and privilege some of the interfaces over others [87, 37, 14]. In this scenario, the interests sent by a user may never reach some of the producers, therefore failing to deliver events.

The last problem in the polling mechanism is related to in-network caching. If a user requests the same content with the same name in the interest (to poll for events), the user may get old events stored in some caches. One way to counter this problem is to add some sequence number in the name. Unfortunately, however, this require some kind of synchronization between consumers and producers, because each consumer needs to know the sequence number of the next event to issue a safe (non-caching) request. This kind of synchronization can be difficult to achieve: if a consumer misses some notifications, the consumer may get stuck in the past and never get newer events.

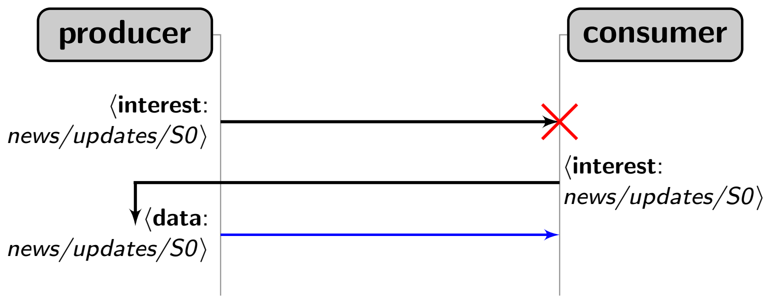

Figure 2.2. Publish/subscribe event notification implemented with interest as notification

The second implementation that we analyze is called *interest as notification* and is illustrated in Figure 2.2. In this implementation the producers notifies all the interested consumers when a new event is available. To do this, the producer sends a special interest that has to be sent in multicast and does not require any data reply. This is the first interest sent by the producer in the picture. At this point, the consumer queries the producer in order to get the new content. This strategy is the most used in CCN. In particular, according to the CCNx 1.0 specifications,[12] an interest with lifetime set to 0 is used as a notification and the requester node does not expect any data in response to this interest.

Using an interest as notification would solve some of the issues of the polling implementation, such as the selection of the correct polling rate, but it would also introduce new problems. One of the main problems is the forwarding of the first interest used as a notification. In fact, there is a clear mismatch between the semantics of an interest and that of a notification. A notification is a multicast packet, which may create loops. In CCN, the PITs are responsible for avoiding loops in the forwarding process of interests. In reality, the PITs may fail to prevent loops in a number of scenarios, as described in Section 3.3.1. In fact, according to Garcia-Luna-Aceves et al., the only way to avoid loops in the forwarding of an interest in multicast is to use a loop-free multi-path routing scheme [37], like the one we propose in the next chapter. Also, sending an interest as a multicast packet is at odds with most of the proposed forwarding strategies where interests are usually given an anycast semantics.

Interests as notifications would also need to be handled differently by the

[12]CCNx Semantics. http://www.ietf.org/id/draft-mosko-icnrg-ccnxsemantics-01.txt

PITs, because they do not require any reply. In particular, the best option would be to never store such interests in the PITs, which would complicate a bit the forwarding process. The other option would be to delete an interest as notification from the PIT at its expiration time, but that would increase the size of the PITs, with the same scalability problem that we already highlighted for the polling implementation.

Polling and interests-as-notifications are not the only ways one could implement push communication in CCN. But they are the ones that require zero or minimal modifications to the original architecture. In the literature there are other examples of publish/subscribe event notification implemented in CCN, but they always require significant changes in the architecture. For example, in COPSS [23] the authors introduce the Subscription Table (ST) in the rendezvous points, which are special nodes able to forward notifications. Another example is KITE [89], where the authors introduce a new kind of interest packet, called tracing interests. These special interests are similar to subscriptions in the publish/subscribe terminology, in the sense that they set state in the PITs of the routers that can be used later by publisher nodes to forward data packets.

In conclusion, we can say that it is possible to implement a push communication primitive on top of the basic CCN architecture. However this leads to additional traffic, packet management overhead, or some significant changes to the original architecture. In the following section we demonstrate that an ICN architecture can support push communication natively without introducing additional data structures and with a minimal effort.

2.1.3 Native Push and Pull Communication API

In the previous section we argued that on-demand content delivery (pull) and publish/subscribe event notification communication (push) primitives are different. In fact, even if it would be possible to implement publish/subscribe event notification over a pull architecture, that may be problematic in practice. However, here we argue that the two communication modes are not *that* different, and more specifically that they have some fundamental common elements. In this thesis we exploit this commonality to implement a forwarding table that is able to support both on-demand content delivery traffic and publish/subscribe event notifications.

First of all we need to analyze in detail all the steps of the communication process in both push and pull flows. In Figure 2.3 we represent the packets exchanged between the producer and the consumer in a pull communication flow. As a first action, the producer sends a *register* packet to the network,

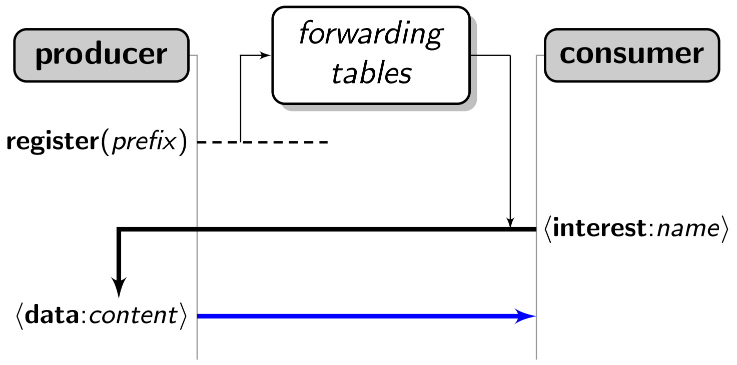

Figure 2.3. On-demand content delivery (pull) packets exchange

specifying the prefix or the name of what the producer wants to publish. With this action, the producer communicates to the network that it can produce the content related to the prefix specified in the register packet. This information is disseminated and processed through some routing protocol, so that each router would use that information to update is own forwarding table. At this point a consumer can request some content with an interest packet with a specific name. The network forwards the interest packet toward the producer using the information stored in the forwarding table of each router in the network. When the interest reaches the producer, or a router with the required content in its cache, a data packet is sent back to the consumer.

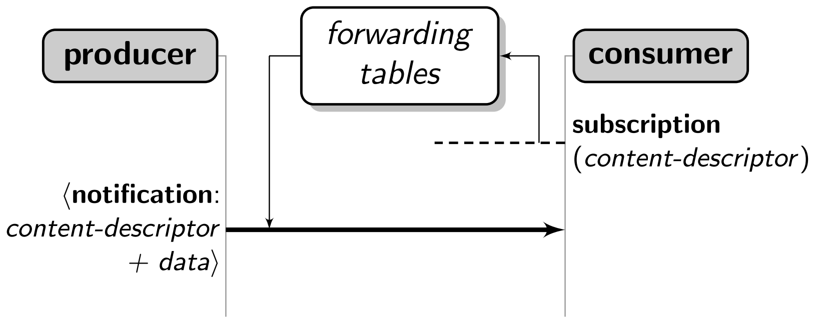
producer
forwarding tables
consumer
subscription (content-descriptor)
⟨notification: content-descriptor + data⟩

Figure 2.4. Publish/subscribe event notification (push) packets exchange

Figure 2.4 describes the phases of push communication. This time, the consumer performs the first action, communicating to the network that it wants to receive some kind of information. Thus the consumer sends a subscription packet, describing the content that the consumer intends to receive. Like the register message in pull mode, a subscription is disseminated and processed in the network through a routing protocol, possibly resulting in routers updating their forwarding tables. When a producer has an event to notify, the producer simply sends a notification packet. The notification is forwarded as a multicast packet using the information on the forwarding tables of each router so as to reach all interested consumers.

In both push and pull communication there are essentially three phases, summarized in Table 2.1:

1. A node generates routing information that feeds into the forwarding tables.

2. Routers forward packets according to their forwarding tables.

3. The forwarding process delivers information to the right users.

In the pull communication model, it is the producer that generates the routing information, whereas in push communication it is the consumers. However, in both cases the register/subscription packets are disseminated and processed by each router, according to some routing protocol, to create or update state in the router's forwarding table. This suggests that register and subscription packets are semantically identical. And indeed their semantics can be shown to be identical, which is why in our architecture we use a single packet that can serve both as a subscription for a consumer (in push) or as a register packet for a producer (in pull).

	Pull	**Push**
(1) Routing Information	Generated by producer Used to update FIB	Generated by consumer Used to update FIB
(2) Matching	FIB	FIB
(3) Information Delivery	Anycast Requires Reply	Multicast One-Way

Table 2.1. Three phases in the communication process: pull vs push

In the matching phase, in both cases, the routers match packets, notifications or interests, against the same forwarding table, and they forward these packets using the same rules. For this reason we use a single forwarding table. The only difference in forwarding between push and pull is the fan out of notifications and interests, respectively, which plays a crucial role in the third phase.

In the third and last phase, the two communication flows differ significantly. As described in Table 2.1, in the case of pull communication, interests are anycast packets, meaning that it is important that an interest reaches *at least one* producer (and preferably only one of them). Conversely, a notification is a multicast packet that must reach *all* interested consumers. This difference is essential and it is the reason why our forwarding table supports different fan-out levels. We will discuss this feature later, but briefly this means that a header in the packet, which differs in interests and notifications, tells the router to limit the forwarding to one or more, or possibly an unlimited number of final destinations.

The other main difference in the third phase is that notification packets (push mode) are one-way messages, while interests (pull communication) require a reply packet that must return data back to the consumer.

In summary, we consider notifications and interests as semantically different packets, and in our architecture we consistently use two different packets.

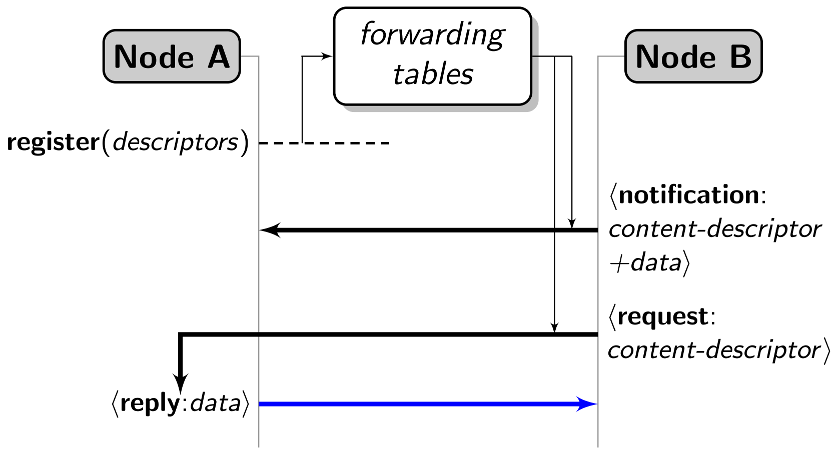
Node A
forwarding tables
Node B
register(descriptors)
⟨notification: content-descriptor +data⟩
⟨request: content-descriptor⟩
⟨reply:data⟩

Figure 2.5. Unified content-based network layer

Figure 2.5 illustrates the architecture we propose. We use only one packet, called *register*, to define and transmit routing state. This distributes the state in the network and creates or updates forwarding tables. We then use the forwarding tables to route both *request* and *notification* packets. Requests are equivalent to an interest in CCN terms. However, we prefer the term request to avoid confusion between immediate interests (requests) and long-term interests (subscriptions). The forwarding algorithm is the same for interests and notifications, the only difference is in the fan-out of the two packets. Then, for each request, the network answers to the user with a *reply* packet that carries the required data, like a data packet in CCN.

Recently, the CCN developers also started to look at push communication. What they concluded is available in the ICNRG draft “Support for Notifications in CCN”.[13] In this document they describe conceptually an architecture that is very similar to TagNet. The developers realize that a push communication is essential for many applications, and that implementing it over pull primitives is inefficient. The developers also decided to use a new packet for event notifications with a different semantic with respect to a normal CCN interest. This packet, called notification, is forwarded using only the FIBs, without leaving any state in the PITs, and without expecting a reply. We consider these recent developments as an excellent validation of our ideas on push and pull communication in ICN [17, 60].

2.2 TagNet Naming Scheme

As we discussed briefly in the introduction chapter (Section 1.3), there are at least two naming schemes in ICN: flat names and hierarchical names. Each one of these naming schemes has its own advantages and disadvantages.

A flat name is easy to match, because it supports exact-match semantics and it can be designed as a “self-certifying” name. However, networks based on flat names do not scale easily, and they require a name resolution system, similar to DNS, that is external to the network, meaning that the packet forwarding is not based on content, and instead still based on network-defined addresses.

Hierarchical names seem to be more promising in terms of scalability for ICN, because they can aggregate by prefix. However, hierarchical names have their drawbacks, too, as they require a more complex encryption mechanism and a more sophisticated matching procedure. In this section we focus on hierarchical names and their implementation in NDN and CCN, because they do not require any external lookup service, which is the same design we adopted for TagNet.

2.2.1 NDN Naming Scheme

In NDN, interest packets carry only a few header fields. Specifically, the only mandatory ones are the name and the nonce, which is a unique identifier of the packet.[14] For this reason, all the information related to the application must be embedded within the name. This information can be of various kinds: for

[13]Support for Notifications in CCN. http://www.ietf.org/id/draft-ravi-ccn-notification-00.txt
[14]NDN Packet Format Specification. http://named-data.net/doc/ndn-tlv/interest.html

example, it could be related to the user sending the interest, like cookies in an HTTP request, or it can amount to an application-specific command.

This design choice—to embed all sorts of information in the name—can be problematic for different reasons. First of all, at each step, the name has to be analyzed in order to understand which part is useful and which part is unnecessary for the current function. This can be particularly problematic in the forwarding process. Most of the implementation proposed so far in fact can achieve high throughput when the number of components of the name is relatively low or is not much larger than the length of the matching prefix in the FIB.

Another problem is related to usability of these names. As highlighted by Moiseenko et al., putting all the information in the name complicates the implementation of a simple protocol like HTTP, which should fit the basic NDN principles of request/reply communication perfectly well [55]. In particular, Moiseenko et al. note that transferring the necessary data from the client to the server using only the interest name is not trivial, simply due to the amount of information that needs to be exchanged. Moiseenko et al. then propose various potential solutions to address this problem. One proposal is to create another field in the packet header to carry information that are specific to the application but do not need to affect the forwarding of the interest.

2.2.2 CCN Naming Scheme

The original CCN scheme was very similar to the one still used in NDN.[15] As in NDN, in the original CCN all the information are in the name.

Since CCNx 1.0, however, interest packets have the same structure of the data packets, so they can carry a payload. This payload can be used to send additional information that is specific to the application but is not needed for forwarding.

Another new feature introduced in CCNx 1.0 is Labeled Content Information (LCI).[16] This amounts to a structured name where each segment or component has a type. An example of LCI is the name: *lci:/Name=com/Name=google/-Name=gmail/App:0=user/App:1=password*. In this name each segment has a label: the label *Name* indicates a segment that is really part of the name, while the label *App:n* indicates a segment that carries some information for the application. Each App segment is also enumerated. A similar approach can be

[15]Some of the changes in the packet format applied in NDN since CCNx 0.7.2 are described on the NDN website. http://named-data.net/doc/ndn-tlv/changelog.html

[16]Labeled Content Information. http://www.ietf.org/id/draft-mosko-icnrg-ccnxlabeledcontent-00.txt

used to retrieve a chunk of a file, like in the name *lci:/Name=ch/Name=usi/-Name=papalini/Name=thesis/version=3/chunk=12*.

Using the LCI encoding, it is possible to decide which segment is useful for which function. The segments labeled *Name* are used in the forwarding process, *App* segments are useful for the applications, while *version* and *chunk* segments can be used by a transport protocol. This implementation can be more efficient with respect to the standard NDN naming scheme, because each protocol can extract the relevant parts from the name and ignore the rest.

2.2.3 Multi-Modal Addressing in TagNet

What emerges from the above discussion on naming, and in particular from the evolution of the naming scheme in CCN, is that using a unique name to carry all the information is not a good design strategy. This is clear also from the first proposal of CCN [47], where the authors present the name as a concatenation of specific fields, similar to the one shown in Figure 2.6.

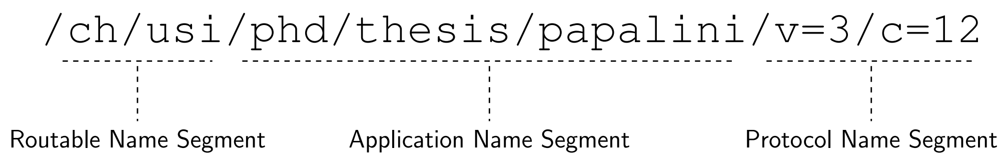

Figure 2.6. CCN/NDN name components

In Figure 2.6 we see a part of the name, called *routable name segment* that is used to route the name across the network. This part represents the location of the object and is essentially a network-defined address. The *application name segment* is instead the application-dependent part that may contain a directory path on a server, like in case of Figure 2.6, or some parameters that are required by the application. The last part of the name, called *protocol name segment*, specifies which version (*v=3*) and which chunk of the object (*c=12*) the name refers to. This part is not useful to the routers for the forwarding function, nor to the user who would not want to deal with such low-level information. It is instead really useful to the transport protocol (to address a specific sequenced chunk) and to in-network caches (to uniquely identify a data chunk).

What we conclude from this analysis is that we need different components in a name. Each component is specific for a particular function and it may be completely irrelevant for others. For this reason, in TagNet we propose to use an addressing scheme with three independent components: a *descriptor*, a *content* *identifier* and a *locator*. Each component can coexist with others in the header of a packet, and none of them is always mandatory. Figure 2.7 represents the header of a packet with all the addresses, and for each one we highlight the functionality for each layer in the protocol stack.

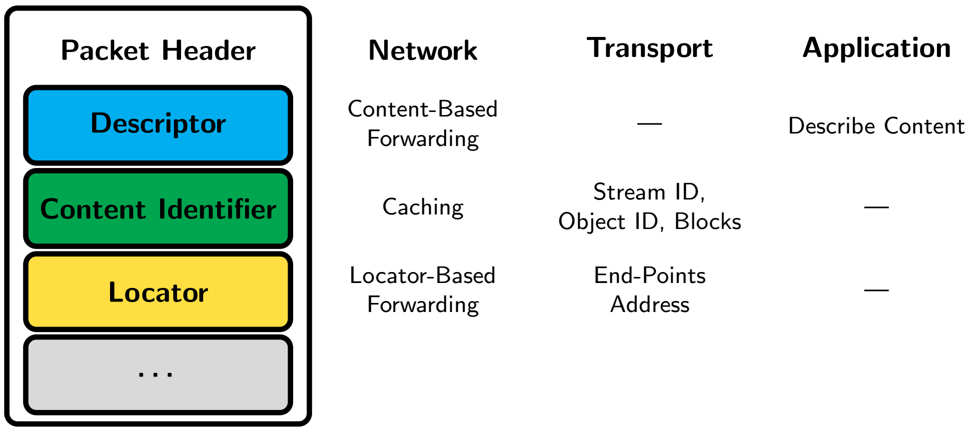

Figure 2.7. Multiple addresses naming scheme with the functionality of each address

In the remainder of the section we describe each type of address, specifying its characteristics and functionalities in the detail.

2.2.4 Application-Level Addressing: Descriptors

In TagNet we prefer to *describe* content rather than naming it as is done in CCN. For that we use descriptors consisting of sets of tags. Application may use descriptors that contain no reference to any network location. Descriptors can be used by consumers to define the kind of information they want to receive, or by producers to declare the content they want to serve.

The idea of describing the content, instead of naming it, is not in itself new. For example, systems such as XTreeNet describe the content using descriptors similar to the ones we propose [30]. The same addressing scheme is reused in more recent works [23, 22]. Publish/subscribe systems also use descriptors, often more expressive than tag sets. As an example that is indicative of a large body of literature on publish/subscribe systems, Carzaniga et al. propose to use predicates to describe the content [20]. Predicates are conditions on the values of typed attributes that describe the content, and they are more expressive than the descriptor we use. In fact predicates amount to tag sets, in the sense that they require a certain minimal set of attributes, but they also allow to specify constraints on the values of those attributes (e.g., numeric constraints such as $price < 100$). For this thesis we adopt tag sets because they seem to balance expressiveness with simplicity and effectiveness. In fact, tags are already widely used in many applications and they are powerful enough to describe content with a good degree of expressiveness.

Descriptors, represented in Figure 2.7 with a blue rectangle, are used at the application level to refer to information or perhaps to application components or users, while at the network level they are the basis for routing and forwarding (as described later in Chapters 3 and 4). Since descriptors have this double functionality, they have to achieve two main design goals. First, they need to be powerful enough to express the interests of applications. Second, they need to aggregate as much as possible to minimize the sizes of forwarding tables in routers.

Descriptors play a central role in TagNet, analogous to IP addresses in the current Internet. A descriptor can be seen as the address of a node in the information space. For example, a content provider that wants to serve the movie *Pulp Fiction* by Tarantino in high definition, could register the descriptor {*Pulp Fiction*, *Tarantino*, *HD*}. This descriptor, effectively, becomes part of the address of the provider. Similarly, a user interested in weather forecasts for the city of Lugano, Switzerland, might register the descriptor {*weather*, *Switzerland*, *Lugano*}. As in the previous case, this descriptor becomes part of the address of that node. A node may provide more than one object, or may be interested in more than one topic. The set of all the descriptors announced by a node forms the address of that node. A set of descriptors is also called a *predicate*.

The matching relation for descriptors is defined as the subset relation between sets of tags. A descriptor D_1 matches another descriptor D_2 if D_1 is a superset of D_2, and so $D_1 \supseteq D_2$. This relation is used to match packets in the network: it describes how request packets match the objects published by the producer nodes, and how notifications match the subscriptions of the interested users. More in detail, a descriptor R in a request matches a publication P announced by a producer if $R \supseteq P$. In the same way, the descriptor N in a notification matches the subscription S of a user if $N \supseteq S$. For example, a request with descriptor {*Tarantino*, *Pulp Fiction*, *HD*, *1994*, *ENG*} would match the publication with descriptor {*Tarantino*, *Pulp Fiction*, *HD*}. Similarly, the notification with descriptor {*update 4:30PM*, *weather*, *Switzerland*, *Lugano*, *sunny*, *temp: 20*} would match a subscription such as {*weather*, *Switzerland*, *Lugano*}.

Descriptors are more general, meaning more expressive, than hierarchical names. Intuitively, a descriptor can be seen as a name prefix in which the order of the of the name components does not matter. Consider a user who wants to download the movie *Pulp Fiction* in English but with Italian subtitles. The user might express this request with the hierarchical names */Pulp_Fiction/Eng/ Ita_Sub/* or */Pulp_Fiction/Ita_Sub/Eng/*. Both are plausible names but they refer to different objects. This ambiguity increases when we add more parameters in the request (e.g., high-resolution, pricing). Generally speaking, in order to define all the addresses expressed by a single descriptors we need an exponential number of hierarchical names, one for all the possible permutations of the tags in the descriptor.

Although the set-based semantics of descriptors is useful and expressive, the order of tags may be meaningful, as in hierarchical names. Fortunately, tag sets are expressive enough to serve applications also in these situations, and in fact a tag set can emulate a hierarchical name or prefix simply by explicitly enumerating the components of the name. For example, the hierarchical name */ch/usi/phds/papalini/thesis.pdf* can be expressed with the descriptor *{1#ch, 2#usi, 3#phds, 4#papalini, 5#thesis.pdf}*. Notice that this representation of a hierarchical name is consistent with the subset matching semantics of TagNet. In other words, TagNet can support both hierarchical names and tag sets within the same FIB.

So far we discussed about the expressiveness of tag sets. The other design goal for descriptors is that they aggregate as much as possible in order to reduce the size of the forwarding and routing tables. Content descriptors based on tag sets aggregate according to the subset relation. Given a descriptor D_1 and a descriptor D_2 , we say that D_1 *subsumes* D_2 if D_2 contains D_1 ($D_2 \supseteq D_1$) , and therefore a router can store only D_1 to represent the two descriptors. This is possible because a packet with a descriptor D that matches D_2 will always match D_1 too, because $D \supseteq D_2 \supseteq D_1$. With a more practical example, the subscription *{weather, Switzerland}* subsumes the subscription *{weather, Switzerland, Lugano}*, because every message that matches the latter would also match the former. Subset aggregation is more general than prefix aggregation, and therefore tag-based descriptors would aggregate at least as much as name prefixes for the same data.

Architectural Role of Descriptors

Descriptors are a central and crucial component of the TagNet architecture. It is therefore important to clearly define their nature and purpose within TagNet. In this section we also highlight some of their limitations and discuss ways that descriptors could be modified in order to improve their expressiveness.

First of all, it should be clear that descriptors have no particular meaning from the point of view of the network. A tag set is encoded within a packet header (as explained in Section 3.2), and it is used, as a set, by the network to make forwarding decisions. However, again from the network perspective, individual tags have absolutely no meaning. For example, for a router, the two descriptors {*Tarantino*, *Pulp_Fiction*, *HD*, *1994*, *ENG*} and {*update 4:30PM*, *weather*, *Switzerland*, *Lugano*, *sunny*, *temp: 20*} are just two different abstract sets, and instead it is the application that formulates and interprets one as a reference to a movie and the other one as a weather forecast.

An other important architectural principle of TagNet is that descriptors are intended to serve as addresses, not as a generic containers for application data. Descriptors should describe content (or components, users, etc.) accurately—as accurately as possible—but they should also be as small as possible. In other words, the purpose of a descriptor in a packet is to allow the network to forward that packet, not to transmit data. Furthermore, the TagNet architecture does not even guarantee that a descriptor in a packet is visible as a plaintext to receivers, so the receivers do not know the tags in an descriptor.

For example, one can imagine that some applications may associate machine-generated tags with some videos and use these tags in the descriptors. This set of tags can be quite large and the subset matching semantic of descriptors fails to work properly in this case. In fact, a packet descriptor with a long list of tags may be forwarded everywhere in the network, because it is easy to find a subset for such descriptor. In contrast, if we use a large set of tags to announce some content, this content may not be reachable, since is really difficult to create a superset of a large set of tags. Some other considerations on this aspect are presented in Section 3.2. For this reason, the machine-generated tags should not be used as descriptor to advertise or forward videos in our application. These machine-generated tags are meta-data associated with the content that are meaningful for the application, but not for the network. The right way to handle these machine-generated tags for videos is to create an application similar to the YouTube website. To implement such application in TagNet we can create descriptors to route packets toward servers that can provide some videos, maybe within a specific section (politics, sports, ...), and then the user can send keywords to the server in order to search for the right video. These keywords are part of the payload of the packet, because they do not define any routable information and they are application specific. In particular, they can be used, within the application, to search in the database of machine-generated tags.

Descriptors are location-independent by nature, but they can also by used to target a specific application or user using some precautions. Before dive into this discussion it is important to clarify that the simple addition of a tag in a packet descriptor that refers to a particular source does not guarantee that the packet will reach the specified source. For example, a descriptor such as {*youtube*, *Pulp_Fiction*} can be forwarded to a YouTube server, but it may also match the content advertised by a server that hosts a blog with a post about Pulp Fiction, since the network may ignore the tag *youtube* in the forwarding process.

However, there are ways to use descriptors as an address. A simple way is to encode the address of the application (e.g. the IP address of the machine where the application is running) as a tag in a descriptor. Using this tag the network can forward the packet to the right place. Unfortunately, the usage of addresses as tags may reduces the chances for aggregation and this has an impact on the routing and forwarding protocol scalability. For this reason, even if possible, this usage of the descriptors should be avoided.

A descriptor used to address a specific user or application needs a particular structure in order to guaranty the scalability of TagNet. First of all, the entire descriptor need to be used only to address the user/application, without mixing this location-dependent address with information about the content. Second, these descriptors need some kind of structure to allow scalability. For example, hierarchical names like the ones used in CCN are a good candidates for this particular purpose. As described in the previous section, descriptors can easily emulate hierarchical names, so we can mix the two types of addresses in the same network.

There are also limitations in the usage of tag sets as descriptors. One of the main limitations is that descriptors can not be used to express exclusions. For example, a user may want to request the usual movie Pulp Fiction with italian subtitles, but also not with 4K definition. Notice that a tag set might express this request quite well, for example with tags {*Pulp_Fiction*, *ita_sub*, *-4K*}, where the "-" sign in the *-4K* might indicate any screen resolution other than 4K. But unfortunately this can not be done with descriptors because the subset matching semantic does not express this concept at the network level. In fact, the network would consider *-4K* as a tag like any other—that is, one element of the universe of tags—and would apply the subset semantics to forward the request, which might mean that the network would simply ignore the to *-4K* tag and forward the request to an application advertising {*Pulp_Fiction*, *ita_sub*}, which might not have the move in a resolution other than 4K.

One way to make descriptors more expressive to address the lack of exclusions is to define some tags as mandatory, meaning that a packet would contain a descriptor and also a mandatory subset of that descriptor. The network would then be required to match the packet descriptor only with FIB entries (tag sets) that contain the mandatory subset of the descriptor. In the movie example, the request would indicate *-4K* as a mandatory tag, and therefore the network would only forward the request to applications that explicitly advertise *-4K* in their descriptors. We considered this more expressive semantics but ultimately decided not to adopt it because of its implications on routing and forwarding.

2.2.5 Transport-Level Addressing: Content Identifiers

Tag sets can store all kinds of information. In particular, a tag set could contain a unique identifier for a particular object, such as a hash of the content of the object with a version number and a chunk identifier. Such a descriptor carrying an identifier could be used to identify a chunk of a particular file uniquely. However, as we already argued in our critique of CCN names and the way they are used to embed information, we consider this form of identification in a descriptor as a bad design for a naming scheme.

The information related to the chuck identifier or the version number are still useful, but they are useful for the transport layer, and are mostly irrelevant for other network functions such as forwarding. In TagNet we therefore decided to add a specific address that uniquely identifies each piece of content. We call this address *content identifier*.

The functionalities of the content identifier are illustrated in Figure 2.7, where the content identifier is represented as a green box. At the network level, the content identifier can be used to retrieve the requested content from a cache, which is both more semantically consistent than names (because identifiers leave no ambiguity as to the desired content) and also more efficient, because identifiers can be used with an exact-match semantics that admits to very fast access. Identifiers can also be very useful for a transport protocol, in particular to request the next chunk of a stream or file.

2.2.6 Network-Level Addressing: Locators

The last component of our naming scheme are locators, indicated in yellow in Figure 2.7. A locator identifies a particular node in the network. The main difference between a descriptor and a locator is that, while descriptors are assigned by applications, locators are assigned by the network. Application-defined addressing is arguably the most important property of ICN, because allowing applications to define network addresses empowers applications to define semantically meaningful information flows, without having to worry about the structure of the network itself. However, at the same time, application-defined addressing leaves the network and also applications themselves vulnerable to conflicts and abuses in the use of the address space. Perhaps more fundamentally, application-defined addressing is inherently less scalable than network-defined addressing.

Locators instead are controlled by the network and can therefore be chosen so as to obtain compact forwarding tables and fast forwarding algorithms. The main purpose of a locator is to forward packets towards a known network destination. In particular, locators can be used in a request/reply exchange to route data packet back to the requester. This way we can avoid to store in-network state for every packet. We also propose to send requests using locators whenever possible so as to increase the throughput of the forwarding algorithm at each node. We will discuss such a request/reply data exchange in Section 3.3.3.

2.3 Evaluation

In this section we evaluate the communication model proposed for TagNet. The purpose of the evaluation is to substantiate the design choices and claims presented in Section 2.1, where we argue that an ICN architecture should provide natively a publish/subscribe event notification primitive together with the on-demand content delivery primitive. In particular we show that with a native publish/subscribe event notification primitive we can have more efficient applications in terms of functionality (missed/duplicated notification), generated traffic, and additional in-network state.

To set up and run our experiments, we create a prototype of TagNet as an extension of CCNx version 0.4.0,[17], which was current at the time we developed these experiments. Unfortunately, the current version of CCNx (version 1.0) is not backward compatible due to some implementation and architectural changes, so we did not run this evaluation on the most recent version of CCNx. However, we still believe that the evaluation and its results are significant and indicative of the CCN architecture in general. Later we discuss the possible implications of the differences between version 0.4.0 and version 1.0 of CCNx (see Section 2.3.1 below).

Also notice that, in order to set up a meaningful comparison with CCN and its forwarding engine, we use workloads with hierarchical names, even though hierarchical names are just a particular specialization of our descriptors.

To evaluate our proposed architecture, we use two types of applications: a file transfer application and an event-notification application. In both experi-

[17]Binary of the current version of CCNx code. http://www.ccnx.org/ccnx-binary-downloads/

ments we compare the CCNx basic implementation with TagNet. The network that we use in our experiments is represented in Figure 2.8. In all the experiments we compute all routers' forwarding information base (FIB) off line. For the CCN experiments we populate the FIB so as to use the shortest path every time. For TagNet we use the routing protocol that we introduce in the next chapter. In particular, we create shortest-path trees from random nodes in the network and we route all the traffic over these trees. When we need to forward an interest, the first router commits the packet to a specific tree, which is chosen in a round-robin fashion, and the packet is forwarded on that tree.

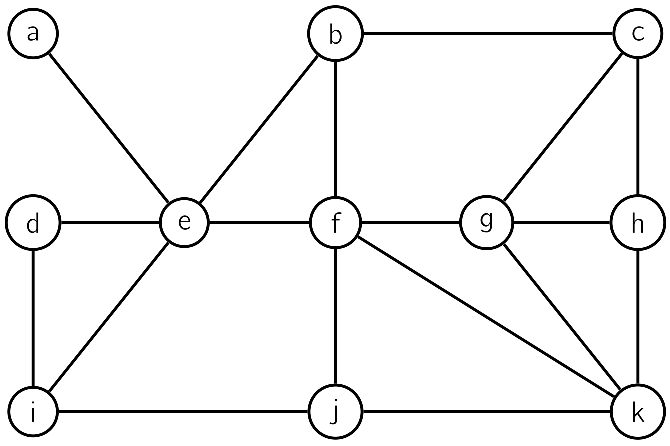

Figure 2.8. Network used in the experiments

For file transfer, we use the application provided with the CCNx 0.4.0 code that can work on both versions of the content router. We run two series of experiments with files of different sizes available from four locations corresponding to routers c, e, i, k in Figure 2.8. In the first experiment we set up a single consumer on router g. In the second experiment we have four consumers on routers a, g, h, j. We repeat each individual experiment 10 times, each time randomly choosing different nodes as roots for the routing trees. This selection may effect the traffic generated by TagNet because some trees may stretch the path between nodes (see Section 3.3.4 for more details).

Figure 2.9 shows the results of the experiments conducted with the file transfer application. We label *F1* the experiments with a single consumer, and *F4* the runs with four consumers. We measure the traffic incurred by the file transfer, measured in the total number of packets crossing a network links. The result shows that the two architectures are almost identical, for both F1 and F4, with respect to the generated traffic. TagNet generates a bit more traffic because some paths in the network may be stretched by the routing scheme.

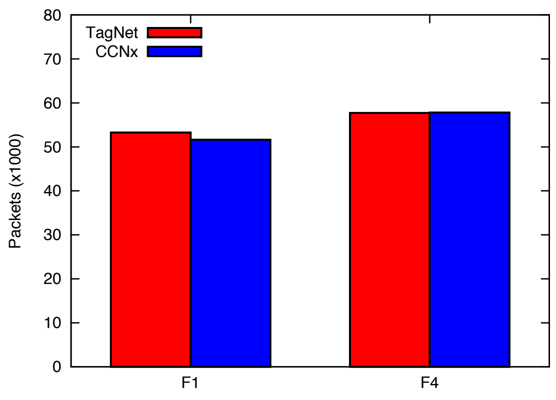

Figure 2.9. On-demand content delivery application: generated traffic

To test the event-notification scenario, we implement the two notification services described in section 2.1.2 on top of CCNx. One application, denoted as *CCNx+polling*, implements the publish/subscribe event notification with polling as described in Figure 2.1. The second one, indicated with *CCNx+interest*, implements the event-notification using an interest packet to notify new events as presented in Figure 2.2. For TagNet we do not need any specific application, since the network itself can forward the notification packets in the right way. As for the file transfer application, we compute the FIBs in CCN using the shortest path from the producer nodes, while for TagNet we use our trees-cover routing scheme. We run two series of experiments, first with a single subscriber at node g, and then with four subscribers at nodes a, g, h, j. The producer nodes are again located at positions c, e, i, k. Also in this case we run a series of 10 experiments for each one of the three implementations.

All these experiments use a simple publish/subscribe workload in which all producers publish notifications at the same rate and all notifications have the same name to which all consumers subscribe. Therefore, all notifications are supposed to go to all consumers. In order to test the three implementations in different scenarios we use three workloads, each one with a different publication rate. We use publication rates that are higher than, comparable to, and lower than the polling rate used by the polling implementation which we set to 0.5 seconds. We run two high-rate series labeled *H1* and *H4*, with one and four consumers, respectively, in which producers publish every 0.1–1 seconds; two medium-rate series, labeled *M1* and *M4*, in which producers publish every 0.5–2 seconds, and two low-rate series, labeled *L1* and *L4*, in which producers publish every 2–5 seconds.

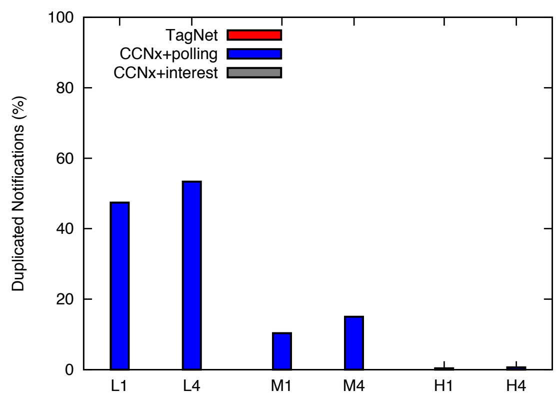

Figure 2.10. Publish/subscribe event notification: duplicated notifications

In the first set of results we look at the application-level effectiveness of each implementation. The results of our experiments are summarized in Figure 2.10 and Figure 2.11. The graph in Figure 2.10 shows, for each series of experiments and for each implementation, the percentage of duplicated notifications received by the subscribers. Figure 2.11 instead shows the number of missed notifications.

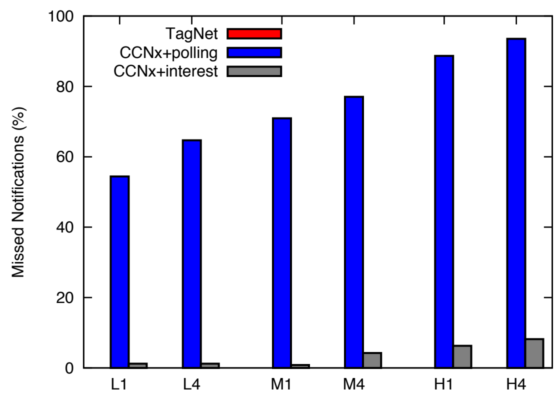

Figure 2.11. Publish/subscribe event notification: missed notifications

The polling application suffers from missed as well as duplicated notifications, with more missed notifications at higher publication rates, and with more duplicates at lower rates. On the other hand, the implementation based on interests has no duplicates and it works almost correctly with low publication rates. However, the interest-as-notification implementation always loses a few notifications, in particular when we increase the publication rate and the number of producers (around 10% of the notifications). The implementation built on TagNet performs well across all workloads, as expected.

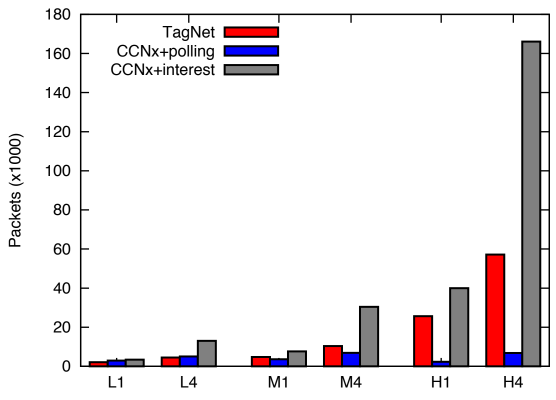

Figure 2.12. Publish/subscribe event notification: generated traffic

In Figure 2.12 we show the traffic generated by each implementation. It is clear from the picture that the interest-based implementations built on CCNx generates more traffic as compared to all other implementations. In particular, it generates almost three times more traffic than TagNet in the experiment with four consumers, which is also the case where the *CCNx+interest* implementation suffers more from missed events. The polling application generates an almost constant traffic that depends on the polling rate. In all cases, TagNet generates less traffic than the *CCNx+interest* implementation.

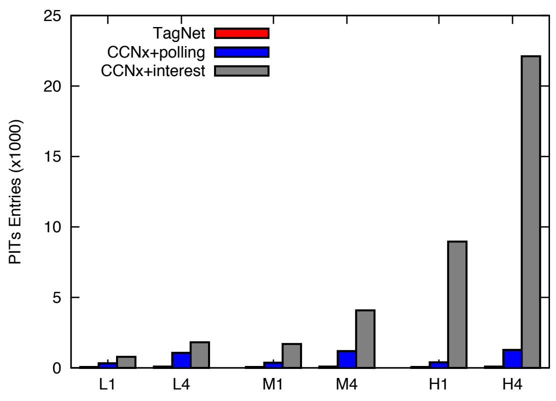

Figure 2.13. Publish/subscribe event notification: in-network state

The plot in Figure 2.13 shows the cumulative in-network state created during the experiments, meaning the number of PIT entries generated by each application. As for traffic, the polling implementation creates an almost fixed number of entries that reflect the polling rate used in the experiment. The *CCNx+interest* implementation generates a large number of entries in the PITs, which is particularly problematic considering that most of the interests have the same name and therefore should be aggregated. This suggests that in a more general case, where the events have different names, the amount of entries in the PITs may grow even more. In the case of TagNet, we do not use the PITs at all, because all the packets are forwarded only using the FIB. For this reason, TagNet does not require any in-network state, as shown in the picture.

2.3.1 Discussion

In this evaluation we used an old version of CCNx, but many new features were introduced in newer versions, in particular starting with CCNx 1.0. Unfortunately, it is not easy to adapt our code to the new implementation of CCNx, so here we try to describe what could be the advantages and disadvantages of the new version of CCN.

One of the new features of newer versions of CCNx is that an interest with lifetime equal to 0 can be used as a notification packet. This can be really useful to improve the performance of the *CCNx+interest* implementation. However, this interest should be handled in a special way that is not specified in the documentation. For example, since there is no need to send a data packet as a reply to these special interests, it is conceivable that routers would not create state in the PITs for such interests. If this is the case, the in-network state in Figure 2.13 can be reduced significantly.

However, a CCNx without PIT entries for zero-lifetime interests may generate even more traffic than the old version we used. This is because some form of PIT entries are still used in CCN to control the forwarding process, not just to return data back to the consumer. In particular, CCN 0.4.0 used a “nonce” for each interest (a unique identifier) to detect loops in the forwarding process. For each interest, each router that receives the interest stores its nonce together with the name in a PIT entry. So, when a router receives the same interest twice (a forwarding loop) the router can safely discard the interest. This process may lead to some problems that we describe in detail in Section 3.3.1. To overcome these problems, CCNx 1.0 uses hop counters to detect and cut loops. In this new implementation, an interest is forwarded until the hop-counter goes to zero, which would prevent catastrophic packet storms but may still produce excessive traffic. Garcia-Luna-Aceves et al. propose a new set of algorithms that can avoid this traffic overhead, at the cost of storing more information in the PITs [37].

Recently, the CCN developers discussed and highlighted some of the problems that CCN has in handling push traffic. The solution that they propose is semantically identical to TagNet: an architecture that can handle both push and pull traffic, using three packets, which in CCN are called interest, data and notification..[18] With the introduction of the notification packets, the performances of CCN and TagNet will be comparable.

[18]Support for Notifications in CCN. http://www.ietf.org/id/draft-ravi-ccn-notification-00.txt

Chapter 3

Routing Scheme

In this chapter we describe and analyze the routing scheme that we developed for our ICN architecture. This routing scheme supports both push and pull communication flows, while compressing the routing state on each node to achieve Internet-level scalability.

The proposed routing scheme is based on trees. We use trees because they are good structures for many routing problems. Trees allow for loop-free paths with simple router-local decisions. Trees also support multicast packets. In fact, we can more generally support different fan-out levels for different packets: we can send a multicast packet, which is the default case for notification packets, or an anycast packet, which is the standard option for request packets, or we can limit the duplication of a packet to a set number of destinations, again using only local decisions at each router. To compensate for the limitations imposed by trees, our routing scheme uses *multiple* trees to cover the network, so as to allow for multiple paths between nodes and better link utilization.

Notice that, although we use our naming based on tag sets to evaluate the routing scheme, the basic principles of this scheme are applicable to other ICN proposals that use different names.

This chapter is structured as follows: in Section 3.1, we present the routing protocols proposed in the ICN literature. In Section 3.2 we describe how we implement the naming scheme introduced in the previous chapter. Section 3.3 describes all the details of our routing scheme and in Section 3.4 we extend the protocol to a hierarchical multi-tree scheme. This version can be used to forward packets both at the intra- and inter-domain level. In Section 3.5 we present the implementation of the routing tables and the algorithm used to process routing updates. In Section 3.6 we combine all the features of our routing scheme to sketch an algorithm that supports both consumer and producer mobility. Finally, in Section 3.7 we evaluate the routing scheme.

3.1 Related Work

The routing problem in ICN has received considerable attention in recent years, as evidenced by the numerous routing schemes proposed in the literature.

NLSR is a routing protocol based on names that uses the CCN packets (interests and data) to exchange routing information [45]. Routers exchange link state advertisement (LSA) packets, like in OSPF, flooding the network. These packets contain information about the network topology (link states) as well as the content available at the nodes. In order to reduce traffic, routers can use the SYNC protocol implemented in CCN to collect updates. Once all the topological information is collected, each router computes multiple shortest path trees, running the Dijkstra's algorithm multiple times. This way, NLSR provides multiple paths to every destination, that is to every node advertising a specific name prefix. NLSR provides also a way to select the best next-hop router for a particular name, using some cost associated to each edge. NLSR may suffer protocol instability due to frequent updates and large forwarding tables. One way to overcame frequent updates is to delegate some of the routing tasks to the forwarding plane [86]. Using different forwarding "strategies", each node can temporary patch a route while the routing protocol proceeds to update the forwarding tables.

A similar approach to NLSR is proposed by Dai et al. [26]. Like in NLSR, this protocol uses OSPF to collect the topology information. The prefixes are announced according to their popularity and only popular names are stored in routing tables. In this work the requests for popular content are forwarded according to the FIBs, while those for unpopular content are broadcast. This approach has the advance of reducing the size of the forwarding tables, since each router stores only the state related to the most popular objects. However the popularity of the content may change rapidly and, when an unpopular content becomes popular, the network can be flooded by a lot of broadcast interests. This protocol also does not support multiple routes to the same destination.

Garcia-Luna-Aceves proposes a routing protocol for ICN based on the distance between a node and the publisher of the desired content, called anchor [35, 36]. Anchors periodically send update packets that trace the number of hops traversed from the anchor. Intuitively, when a router receives an update, the router forwards the update only if it improves the best known distance to the anchor nodes. Therefore, each node knows only the closest anchor for each content name. This routing protocol, similarly to the one that we propose in this chapter, guarantees loop-free routes to all, some, or any of the anchor nodes, according to the needs of the application.

Some proposals try to address the content in the caches at the routing level. Eum et al. propose to advertise the cached content locally, in order to attract relevant interests [29]. The main problem with this work is that, although the content available in the caches is advertised only locally, a frequent cache update may generate high volumes of control traffic. Also, this routing scheme may suffer from reachability problems, since the FIBs may point to content that was already evicted from a cache. This happens when the routing protocol does not react quickly to cache updates.

Saino et al. propose a hash-routing scheme to steer the interest packets toward a cache that may contain the required content [71] . The authors show that they can increase the cache hit ratio using this technique.

Papadopoulos et al. [59] propose two greedy forwarding algorithms in a hyperbolic space, meaning two particular address spaces, to which names must be mapped somehow, that allow for greedy routing. These algorithms can be used as a routing scheme for CCN. Although this routing scheme is promising in theory, it is not clear if it usable in practice. In order to use this routing scheme, one must construct a proper hyperbolic space for addresses that abstracts the network topology, and then map names onto this space. However, there is no evidence that such a space can be constructed in all cases, and the mapping between names and the address space is also problematic. In fact, this mapping must be somehow recomputed for each routing update, and also the network topology and the names have to follow the same distribution, otherwise paths may be stretched significantly.

The last approach proposed for routing in ICN is to use Distributed Hash Tables (DHT) organized as an overlay. DHTs are used as a lookup service that maps the name of a content to its location, similarly to how DNS maps URLs (partially) to IP addresses. Examples of DHT are the name resolution network in PURSUIT [77] and the resolution nodes in SAIL [2]. α-route is another DHT scheme that targets the CCN architecture [3].

3.2 Naming Scheme Implementation

To evaluate our routing protocol we need to implement all the addresses that we introduced in the previous chapter, namely descriptors, locators, and identifiers.

Identifiers have a marginal role, if any, in routing, so in this section we present only content descriptors and network locators. In fact, for the content identifiers we do not have a real implementation so far, since in this thesis we mainly focus on the routing and forwarding, and because the congestion controller developed in in Chapter 5 is mostly based on CCN, and therefore uses hierarchical names. However, we imagine that a hash value computed over the content of the packet, plus a sequence number, could be used as a content identifier. We leave the detailed design and implementation of identifiers for future work.

3.2.1 Descriptors Implementation: Bloom Filters

A content descriptor is simply a set of tags, which are themselves strings. We represent these sets of strings with a bloom filter [8]. Bloom filters allow us to represent every content descriptor with a fixed-length field that hides the tags and reduces the size of their representation. The matching relation between descriptors (sets of tags) remains the same between bloom filters: a bloom filter B_1 matches a bloom filter B_2 if $B_1 \supseteq B_2$. In other words, B_1 matches B_2 if all the bits set in B_2 are also set in B_1. We will talk more about matching in the next chapter.

In order to use bloom filters, we need to choose some basic parameters. The main parameter is the width of the filters, which determines the precision of the matching relation, since $B_1 \supseteq B_2$ may be a false positive, and also the complexity of routing and forwarding. To choose a good width, we imagine that a tag set would most likely contain no more than 15 tags. There are two main justifications for this limit. First, a descriptor is a way to describe an object that a user wants to retrieve. This is more or less what happens when a user searches something using a search engine on the Web, where the search terms are analogous to tags. Looking at the relevant statistics for Web search,[1] we find that most users input only one or two words for their on-line searches, and there are almost no searches with more than 10 words. Therefore, even considering rich applications that add tags to pure user tags, we believe that 15 tags can be a conservative engineering choice that gives enough freedom to users express their interests.

The second justification is that a descriptor with more than 15 tags may become too selective, or not selective at all, depending on the usage. If a user puts too many tags in the descriptor P used to publish some content, there will

[1]Statistics on the number of words per search. http://www.keyworddiscovery.com/keyword-stats.html

be a low chance for the descriptor R in a request to match P, because, in order to match, R has to be a superset of P. The same happens when a user sets a subscription with a descriptor S. If S contains too many tags, there may never be a notification N matching S. On the other hand, if the descriptor in a request or a notification contains too many tags, it has high chances to match a lot of publications or subscriptions. In this case, a user may get unrelated content, or the network may be flooded by useless notifications.

In summary, we engineer our Bloom filters by considering a maximum of 15 tags per descriptors. Therefore we use Bloom filters with $k = 7$ hash functions and width $m = 192$. In this setting, a subset check $S_1 \subseteq S_2$ would be considered true wrongly, meaning resulting in a false positive, with probability

$$(1 - e^{-k|S_2|/m})^{k|S_1 \setminus S_2|}$$

For example, for a descriptor S_2 that contains 10 tags ($|S_2| = 10$), and a descriptor S_1 that differs by 3 tags from S_2 ($|S_1 \setminus S_2| = 3$), a test $S_1 \subseteq S_2$ would be evaluated true, and so results in a false positive because S_1 contains 3 elements that are not in S_2, with probability 10^{-11}.

Therefore, notice that the limit of 15 tags is not a hard limit for the user, who could easily define descriptors of more than 15 tags, with the only problem that those descriptors may be more susceptible to false-positive matches.

3.2.2 Locators Implementation: TZ-labels

As described in Section 2.2, a packet can also carry an explicit destination locator. This can be easily implemented using IP addresses. However in our case we decide to replace IP addresses with TZ-labels. A TZ-label is an address defined in a routing scheme for trees by Thorup and Zwick [76]. We use TZ-labels because our routing scheme is based on trees, and, in this setting, TZ-labels are more efficient than IP addresses. In the TZ routing scheme, each router is identified with a short label, namely its TZ-label. Using the TZ-label of the destination, carried by the packet, and its own TZ-label, a router can take a forwarding decision and send the pack to the next-hop. In other words, a router's TZ-label serves both as the locator of that router and its forwarding table.

This scheme is extremely efficient both in space and time. Each router has to store a single TZ-label, which requires at most $(1 + o(1)) \log_2 n$ bits for a network of n nodes. This efficiency is not just theoretical. In practice, using large networks such has the AS-level Internet topology, we need only 46 bits in the worst case. In terms of time, the forwarding decision on such TZ-labels can be computed by a router in few CPU cycles, resulting in a very high throughput also on simple, general purpose machines. A more detailed description of the TZ-label assignment and the forwarding process based on these locators is available in Section 4.5.

3.3 Routing Scheme

In this section we introduce our routing scheme based on multiple trees. At the core, this is a simple content-based routing scheme on a spanning tree. We extend this basic scheme using multiple trees within the same domain and across network domains. The usage of multiple trees reduces the paths stretch and traffic load on the tree edges. In this section we also introduce the usage of network locators and we describe how to use them to efficiently handle communication flows.

3.3.1 Why Should We Route On Trees?

In this thesis we decided to create a routing scheme based on trees. Trees are good structure to use for routing, as they do not have loops and they can be used easily to forward messages in multicast. These are good properties since ICN is intrinsically multicast. When we have multiple sources for the same content, or when we have multiple subscribers for the same event notification service, there are points in the network where packets need to be duplicated and sent to multiple destinations. This process can easily generate loops.

CCN uses tables called Pending Interest Tables (PITs) at each node. PITs are used to collect soft state related to the interests forwarded by the router but not yet satisfied (and therefore removed) by a corresponding data packet. The PITs have three main purposes:

1. The state in the PITs is used to route the data packets back to the the user that sent the interest.

2. PITs are used to aggregate interests for the same content, and therefore reduce traffic in the network.

3. PITs can stop loops in the interests forwarding process.

In this section we present some scenarios where the PITs fail to detect loops and they lead to a misbehavior of the network. The most recent versions of

CCN (since version 1.0) use a different implementation of the PITs. NDN still uses a PIT similar to the one proposed in the initial implementation of CCN. We therefore compare the behaviors of NDN and the older version of CCN, both indicated with CCN 0.x, with the new CCN, indicated with CCN 1.x

In CCN 0.x, each interest has a name associated with a nonce that uniquely identifies that particular interest. The PIT stores the nonce with the name, in order to avoid to re-send the same interest in case of loops. CCN 1.x does not use nonces. Instead, an interest has a hops-to-live counter that limits the interest transmission. At each hop, the hops-to-live counter is decreased and, when it reaches 0, the interest is dropped. This approach does not prevent loops, it simply mitigates the problems induced by loops.

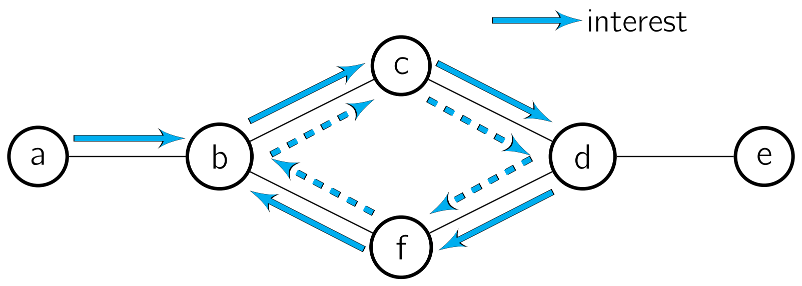

Figure 3.1. Scenario 1: loop path

In Figure 3.1 we show a simple scenario where an interest, represented with a blue arrow, falls into a loop. The interest goes from node a to node b, then b decides to forward the interest along to c, and node b also records the interest in its PIT. In the case of CCN 0.x, this PIT entry contains the name of the interest and its nonce. When b receives the interest for the second time, from node f, it detects a loop, because the interest has a nonce that is already in the PIT. In this case node b discards the interest. However, if the time required for the interest to go through the loop to reach node b for the second time is higher than the timeout set in PIT entry at node b, then the interest may loop. The loop is represented in the figure with dashed arrows. In CCN 1.x, node b is unable to detect the loop, so the interest loops but is eventually discarded as soon as the hops-to-live counter goes to zero. Although this version has the advantage of eventually cutting loops, it may also generate useless traffic, especially if the initial hops-to-live counter is high. On the other hand, if the hops-to-live counter is too low, then the interest may never reach any of its destinations and the user may not receive the requested content.

Other problems with the PITs may occur when an interest is forwarded on multiple paths, as described in Figure 3.2 and Figure 3.3. This happens, for example, when an interest is used as a notification packet. In Figure 3.2, node b sends the same interest to nodes c and f, and the two copies of the interest are propagated to node d. In CCN 0.x node d detects the duplicate interest, since it receives two interests with the same nonce form two different interfaces. What happens in this case is that the second interest, say the one from node f, is discarded and node d keeps track only of the interest coming from node c. The data packet related to the interest goes back to node a using the path d, c, b. This creates a dangling PIT entry at node f that will eventually timeout and therefore will eventually discard. In the case of CCN 1.x, since node d does not detect any duplicate delivery of the same interest, node d will aggregate the two interests from nodes c and f. Node d then forwards the data packet to both c and f, so node b will receive duplicated data (represented with a dashed line).

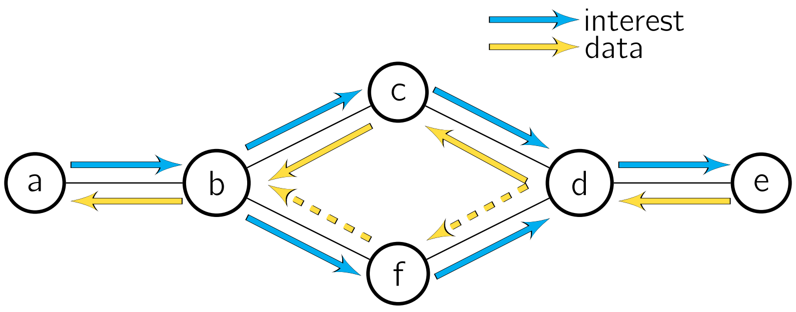

Figure 3.2. Scenario 2: duplicated interest

Ultimately, node b forwards only one copy of the data to a, since the first forward to a also removes the corresponding PIT entry. In this scenario everything works correctly from the application viewpoint, even if in the case of CCN 1.x we have some extra traffic.

However, the same scenario may create some problems, as described in Figure 3.3. As in the previous case, node b sends the same interest to both c and f, and d receives first the interest from c and later the one from f. In CCN 0.x node d discards the interest from f, while in CCN 1.x the two interests get aggregated. At this point, node g sends another interests with the same name, indicated in the picture with a green arrow, to node f. Node f has an entry for that interest in its PIT, so the interests is aggregated. This happen in both CCN 0.x and CCN 1.x. However, in CCN 0.x d forwards the data packet only to node c, because the interest from f was discarded. So, in this case, node g never receives the requested data.

This does not happen in CCN 1.x, because node *d* aggregates the interests that are coming from *c* and *f*, without discarding them, and node *g* receives the required data packet. The data packets sent by CCN 1.x are represented with dashed arrows in Figure 3.3. Also in this case CCN 1.x generates useless traffic, since the data packet from node *f* to *b* is a duplicated packet.

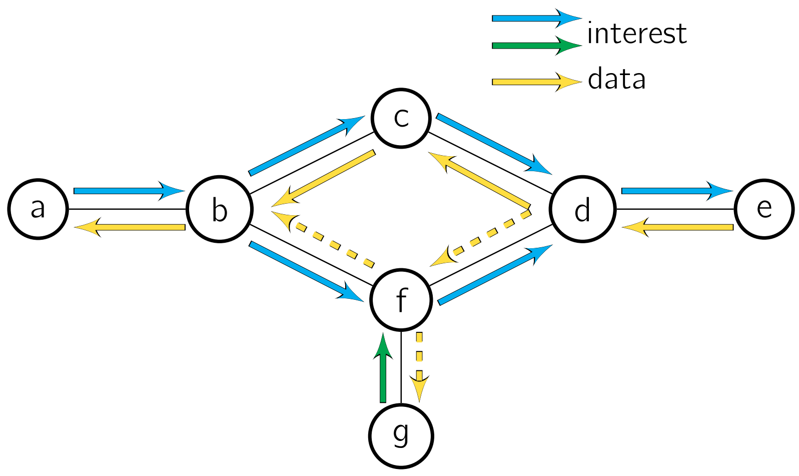

Figure 3.3. Scenario 3: duplicated interest

In a network that is multicast in nature, it is not easy to avoid loops. PITs are generally good to detect loops, but they may fail in some cases. When an error in a PIT occurs, some users may not receive the data that they ask for. Adding the hop counter solves the problems of the PIT, but it introduces traffic overhead.

Our design avoids all these problems, by construction, using trees as a basis for routing. This way packets will never fall into loops, whether they are unicast, anycast, or multicast packets. And furthermore, this design does not require any in-network state thanks to locators.

Other architectures such as PURSUIT propose to use trees to forward packets in what amounts to a multicast group. In particular, thanks to the LIPSIN construction, the PURSUIT architecture can efficiently represent and forward packets along a tree connecting a producer to all consumers [48]. A LIPSIN tree is represented as a set of edges represented with a compact Bloom filter. The notion of a LIPSIN tree is limited to a local domain, where it is reasonable to assume knowledge of all links.

Although based on trees, the PURSUIT architecture and its routing scheme differ quite substantially from TagNet. PURSUIT uses a DHT overlay to locate the content, and then uses a "topology layer" to build and return a LIPSIN tree to distribute content. Thus PURSUIT is more akin to a connection-oriented network than a datagram network, in the sense that an all-knowledgeable and conceptually centralized topology layer must set up a LIPSIN tree before that the tree can be used to transmit content. By contrast, in TagNet we use a set of base trees that are not specific for any producer, and then we forward each packet along one of those trees making hop-by-hop forwarding decisions based on the packet content (descriptor).

It is worth noticing that, since we do not need to keep any in-network state for request packets, we can completely remove the PITs from our architecture. This reduces complexity but also eliminates the aggregation of concurrent requests, and therefore may increase traffic. However, our architecture still allows for in-network caching, which can play a similar role as the PITs for interest aggregation. This is because, as soon as a request is satisfied along a path, all the following requests that cross that path can be satisfied by the cached content. Therefore we believe that our architecture would not introduce much additional traffic, as compared with CCN, even without the PITs. Our intuition is also confirmed by a study conducted by Kazi et al. that shows the dynamics that relate the behavior of the caches and the PITs [49]. In particular, the study shows that, under normal traffic conditions, interest aggregation in the PITs reduces traffic only marginally, and that the majority of traffic reduction is due to cached content. Interest aggregation in the PITs becomes more effective only in the case of heavy loads, when cached content is too quickly overwritten by new content.

3.3.2 Routing on a Single tree

TagNet uses trees as a basis for routing. This means that the network commits each new packet to a tree, identified with a packet header, and then forwards the packet on that tree. We now describe how we route on a single tree. First of all, we need to construct a tree T that spans the network, but for now we assume that we have such a tree. Consider as an example the simple network of Figure 3.4. Each router v knows the list of neighbors adj_v^T that are neighbors also on the tree T. For example, in Figure 3.4, node f knows $adj_f^{T_{green}} = \{b, j, k\}$.

Each router also stores a Forwarding Information Base (FIB) that associates each neighbor w in adj_v^T with a predicate. A predicate p is a conceptually a list of descriptors. In the FIB the router stores a list of predicates $P_{t,w}$ for each neighbor w in adj_v^T. Each list $P_{t,w}$ includes all the predicates announced by all the nodes reachable through neighbor w, including w itself. Figure 3.4 shows the FIB of node b. For example, $P_{T_{green},c}$ is the union of the predicates announced by the nodes reachable through c, namely p_c, p_g, and p_h.

FIB router b	
tree T, next-hop w	predicate $P_{T,w}$
T_{green}, c	$p_c \vee p_g \vee p_h$
T_{green}, f	$p_f \vee p_j \vee p_k$
T_{green}, e	$p_e \vee p_a \vee p_d \vee p_i$

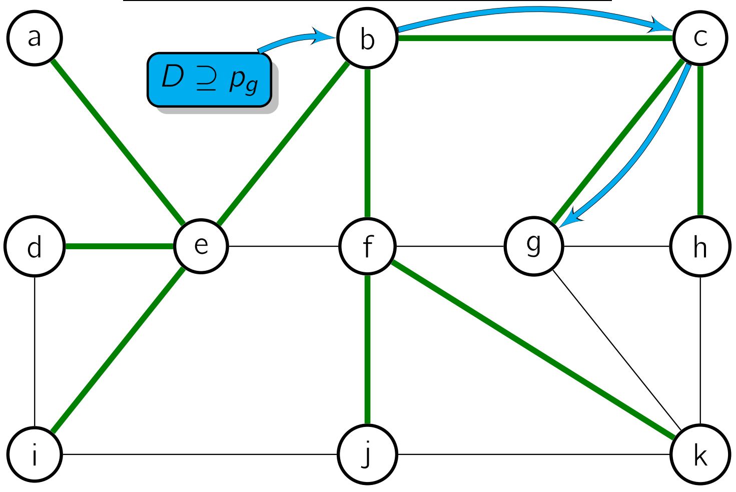

Figure 3.4. Routing on one tree

When a router receives a packet with a descriptor D, it forwards the packet to all the neighbors w in adj_w^T that are associated with a matching predicate. In order to avoid loops, the router never forwards the packet on the incoming interface. We say that a descriptor D matches a predicate p if D matches at least one of the descriptors listed in the predicate p. In the example of Figure 3.4, router b receivers a packet with a descriptor D that matches the predicate p_g, announced by node g. Using the first entry of the FIB, router b forwards the packet to node c. At the next hop, node c sends the packet to g.

Since we route packets on a tree, we can easily control the global fan-out of each packet using router-local decisions. The fan-out k of a packet is a parameter set in a packet header. If we set $k = 1$, the network never duplicates the packet, and instead forwards it to a single destination. This corresponds to a an anycast packet, which is the default setting for a *request* packet. When we set the fan-out $k = \infty$, the network sends the packet in multicast, so each router forwards it along *all* matching interfaces, and the packet will reach all the destinations with a matching predicate. This is the default fan-out for a *notification* packet. We can also set a limit to the fan-out, with $1 < k < \infty$. Using only local decisions at each router we can guarantee that the forwarding process will deliver no more than k copies of the packet. Each router selects at most k output interfaces, corresponding to k neighbors, and then partitions the fan-out over the forwarded copies of the massage.

As described earlier in this chapter (see Section 3.2), we also provide a unicast service on top of our spanning trees. To implement such a service we use locators, and more precisely we use TZ-labels [76]. Using such locators we are able to forward every packet in a unicast fashion extremely efficiently. In the next section we describe how we combine descriptor-based routing and locator-based routing to implement a request/reply service, and how we can exploit locators in the communication flow.

3.3.3 Request/Reply Service and Communication Flow

Every packet in our architecture can carry different kinds of addresses, each with a specialized functionality. With these addresses we can implement a request/reply service that can use expressive and rich descriptors, and, at the same time, use fast forwarding that do not require any in-network state to route reply packets back to the user. The main idea is to forward a request using descriptor-based forwarding, so that a request can be delivered to one or more producers that are able to satisfy such request, but then to use the TZ-label of the consumer to deliver the data back to the consumer.

Figure 3.5 shows a simple request/reply exchange between nodes i and c. The request issued by node i contains a descriptor D that in this case matches predicate p_c, plus the locator of the source node TZ_i in the *src* header. A request packet, as well as a notification packet, may also contain a payload body. The network then forwards the request packet toward a producer of the requested data (in this case node c) using content-based forwarding, indicated with the blue arrows, matching descriptor D against each router's FIB.

When the request arrives at producer c, the producer creates a reply packet that contains a content identifier (shown with a green box labeled Obj-ID in Figure 3.5), the destination locator TZ_i (in the *dst* header) that the producer copies from the source locator in the request packet, the source locator TZ_c, and the requested data. The network then forwards the reply packet to node i using the given destination locator TZ_i, using locator-based forwarding, indicated with yellow arrows in Figure 3.5. At this point, node i knows the locator of the producer node, c, as well the content identifier of the next block of content that it needs to retrieve, which should be derived from the identifier of the content just received. Thus, i can send subsequent requests directly to the producer node c using the fast locator-based forwarding, and also using a content identifier that may be used to retrieve the data from the cache of some router along the forwarding path.

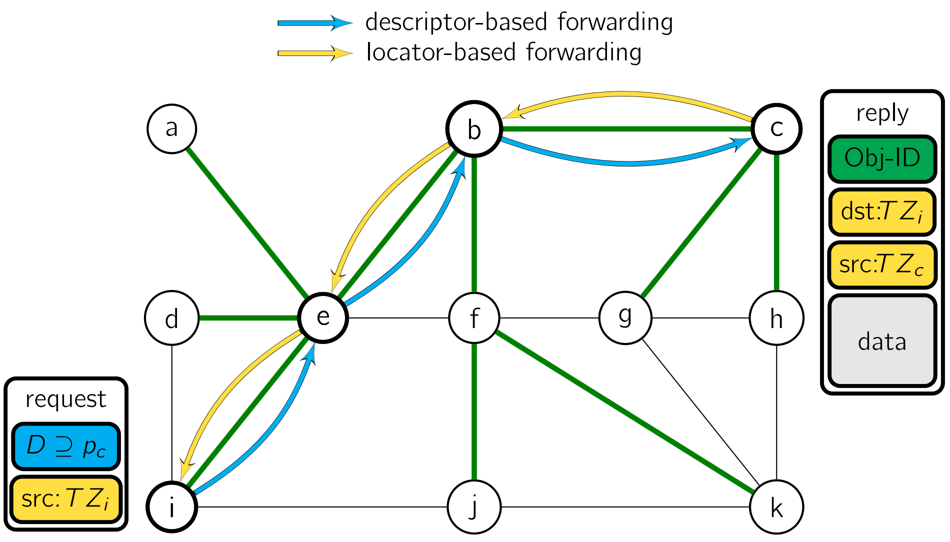

Figure 3.5. Request/Reply Service

Notice how TagNet speeds up forwarding using locators. In particular, notice that the very fast locator-based forwarding applies to most packets in bulk data transfers such as the multi-media streams that make up most of today's Internet traffic. Locators are network-defined (opaque) quantities, and can considered quite stable, much like IP addresses. Still, it may happen that a locator changes during a communication flow, due to the mobility of the nodes involved in the communication. In this case, the transport protocol must reestablish a valid locator, which can usually be done efficiently in the most common case in which only one end-point moves at any given time. We briefly discuss mobility in TagNet in Section 3.6. Another potential problem with locators is related to privacy, since a locator may reveal the identity of a consumer requesting data. If anonymity is required, than a locator should be created so as to preserve anonymity of users, like in onion routing schemes [40].

3.3.4 Using Multiple Trees

So far we discussed about how to route packets on a single tree, using descriptors and locators. Although a tree is a good structure to route packets, it also has two disadvantages illustrated in figures 3.6 and 3.7. Figure 3.6 shows that a tree may stretch paths, meaning increase the number of hops that a packet needs to traverse to reach a destination as compared to traveling on the full network graph. In the example network of Figure 3.6 a packet could go from i to j in one hop using the full graph (blue arrow), but it needs four hops on the tree (orange arrows). This leads to additional latency and traffic.

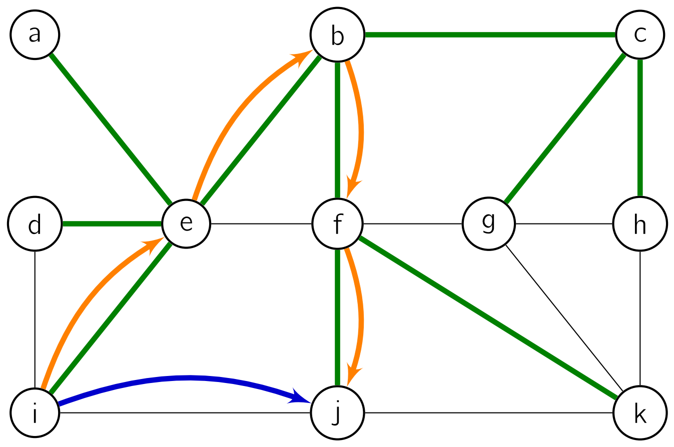

Figure 3.6. Problem using a single tree: higher latency

Figure 3.7 illustrates the second problem, namely increased congestion. Routing on a tree forces all network traffic to go through a limited set of edges, thereby reducing the throughput of the whole network. Consider for example the traffic flowing between the right and left side of the network of Figure 3.7. This is the traffic crossing the cut indicated with the dashed vertical line. Using the tree, all this traffic mush go through the (e, b) link, denoted with the orange arc. However, if we consider the entire network, there are three links that cross the cut and therefore that could carry the crossing traffic. These are highlighted in blue and they are (e, b), (e, f), (i, j). Thus, informally, we can say that using the tree multiplies the congestion on link (e, b) by a factor of three for the traffic crossing the indicated cut. More generally, depending of the topology and on the traffic patterns, using a tree may significantly reduce the overall network throughput.

The stretch and congestion induced by trees are well-known problems that have been thoroughly analyzed from a theoretical perspective. What is also well known, is that it is easy to mitigate these problems significantly by introducing multiple trees. Using multiple trees means setting up multiple trees and then forwarding each packet to one of them. This way it is possible to reduce the stretch and increase the throughput of the network *on average*.

Consider for example the network and tree of Figure 3.6. We could construct another tree that includes the (j, i) link. As we already observed before, the distance between j and i is four on the tree in Figure 3.6, but is one in the new additional tree. If we spread the traffic evenly over the two trees, then, in expectation, the distance between j and i is now 2.5. In general, increasing the number of trees and therefore using more network edges, we can reduce the expected distance and latency between two nodes. The same reasoning holds also for the throughout of the network: multiple trees can be used to increase the throughput and reduce the load on single edges.

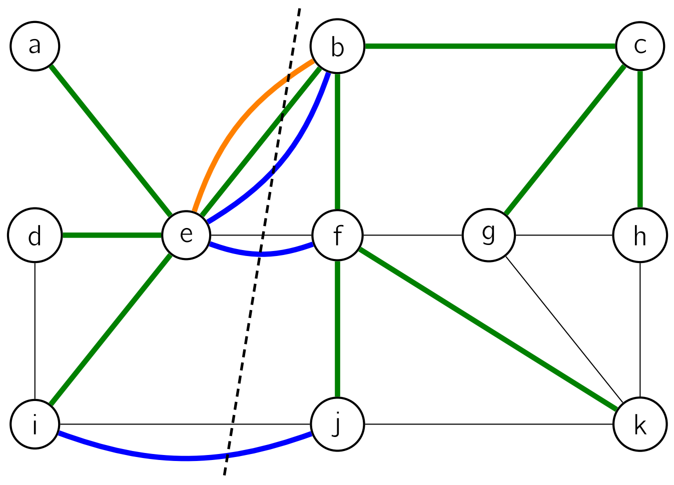

Figure 3.7. Problems using a single tree: lower throughput

The problem of covering a network with trees so as to achieve specific design objectives has been studied extensively from a theoretical perspective. For example, Räcke formulates a method to cover a network with overlay trees to achieve the theoretically minimal congestion under unknown traffic [65]. However, such result does not seem to have an immediate practical applicability. The proposed algorithm produces a very high number of trees, in the order of the number of edges in the network. Too many trees are then complicated to build and maintain. Another problem is related to the algorithm itself, that is centralized and quite complex, and, therefore, would be usable for local trees but definitely not for global trees, meaning trees that span multiple ASes.

Still, we believe that in practice, within a realistic network topology, it is possible to achieve a good compromise between the number of trees and the network coverage. As we will see later, our experimental evaluation confirms this intuition.

Since we use multiple trees, we need to extend our routing algorithm in order to deal with them. For the content-based part we just need to commit a packet to a single tree, somehow identifying the tree within the packet header. Each router has a FIB that contains the information related to all the trees, so it has an entry for each tree T that covers the network, and for each neighbor w in adj_w^T. The router then simply matches the packet against the entry that are related to the tree indicated in the header of the packet.

Locators are similarly easy to adapt. Now a network locator must identify the tree in which it applies in addition to the the TZ-label of the node. In other words, each node now has multiple TZ-labels, one for each tree.

Thus using multiple trees does not pose any serious conceptual issue for forwarding, as the forwarding reduces to the forwarding on a single tree once the tree is properly identified. However, using multiple trees does raise some immediate scalability issues, since each node now needs to store what amounts to multiple FIBs, one for each tree. Later, in Section 3.5, we discuss how to store all the required information, how to compress it, and how to update it. We show in the evaluation section that our routing scheme scales both in terms of memory requirement and update time.

Another fundamental problem is how to build and maintain a good set of trees: one that would be as small as possible, and yet would approximate the full network as closely as possible in terms of path lengths and overall throughput. In this thesis we do not address these topological questions, which are part of another dissertation [60, 61]. Still, for completeness, in our evaluation we show a few preliminary results that demonstrate that even a few trees can approximated quite well the Internet's AS-level topology [16].

3.4 Hierarchical Multi-Tree Routing

In the previous sections we described the routing scheme in a local network, inside an AS. In this section we discuss how we can extend our routing scheme to route packets also among ASes, and therefore define a routing scheme that can be used at the Internet scale.

In our routing scheme we can have multiple trees at different levels in the network, and this creates a hierarchical routing scheme. In this section we describe a 2-level scheme, but it is possible to generalize the scheme to more levels. In our 2-level routing scheme we have multiple trees that are used to cover the AS-level topology, and also multiple trees inside each AS. The first set of trees is used to route packets among ASes for inter-domain routing, while the other trees are local trees, known only by the routers inside an AS. These trees are used for intra-domain routing.

As for the basic case of a multi-tree scheme, each tree generates its own FIB, and, theoretically, each router needs to store all the FIBs related to all trees. In practice, we have a way to compress the routing tables, therefore the FIBs, across trees. In the end, each router has a single FIB that contains all the aggregated information. We describe the aggregation process in the next section.

In the network there are two types of routers: internal routers and gateway router. The internal routers are the routers that are connected only with routers inside the same AS. Gateway routers are the routers responsible for the connection among two or more ASes. Internal routers need to know only the local trees, and they store the aggregated predicates of the hosts that are inside the same AS. Internal routers also need to know at least one TZ-label of one gateway router for each global tree. A gateway router needs to store more information. The FIB of a gateway router contains the aggregated predicates announced by all the ASes. They also store information for all the local and the global trees. A gateway router knows the connectivity with the neighbor ASes, and, in particular, the gateways to use to forward a packet to a certain AS. For this reason, the gateway routers store all the TZ-labels of all the local gateway routers.

Using all this information the network is able to forward packets inside and across different ASes, using a combination of content-based and locator-based forwarding. Here we describe the algorithms used to forward a packet, and later we present an example.

We start describing how we forward a packet using content-based forwarding. A packet is assigned to a local tree by the access router. The packet is forwarded locally on the selected tree, matching the packet against the FIB stored on each router. If the packet needs to be forwarded also at the global level, it is assigned to a global tree. To assign a global tree to a packet the access router sends the packet to a local gateway router, using the locator (or TZ-label) of the selected gateway. In fact, the local router does not have any knowledge of the global trees. At this point the gateway has all the information to route the packet at the global level. It may happen that the packet needs to be forwarded to a neighbor AS that is reachable only from another gateway router, and so, the packet needs to cross the local AS to reach the proper gateway. This can happen because at a high level the AS is a single node, but, in reality, each AS is a network, and the responsibility to route a packet to the neighbor ASes is shared among multiple gateways as described in Figure 3.8.

On the left side of Figure 3.8 there is an AS (gray circle), crossed by two global trees. The picture on the right shows the zoom inside the AS, and it shows the global trees, as well as the local tree (indicated in black) that covers the local AS network. In the figure, only one gateway router participates to both the global trees, while the other two knows only one of the two trees.

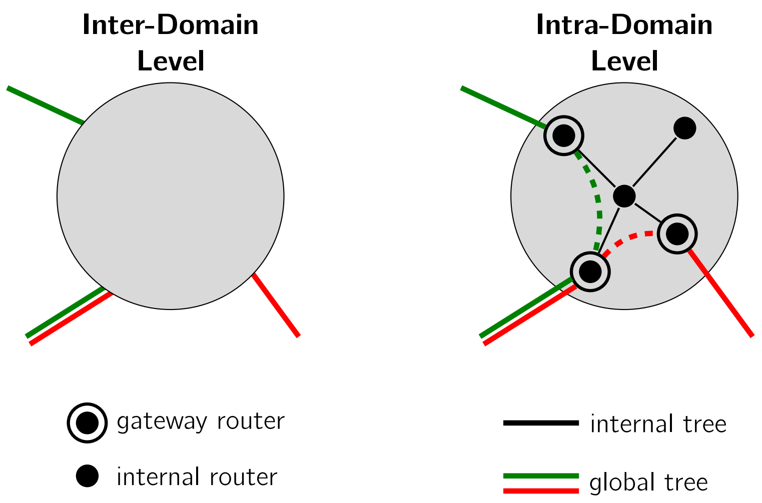

Figure 3.8. Gateway routers and global trees

To forward the packet to the right gateway router we use again the TZ-labels. The same situation may occur when a packet arrives to a new AS that is not its final destination. In this case the packet needs to cross the AS, reach the correct gateway, and be forwarded to the next AS. In this process, the packet is forwarded using locators on a local tree.

The other way that we can use to forward a packet is to use locators. In the hierarchical extension of our routing scheme each locator is composed by a stack of node locators (that can also contain a single value), and, as in the multiple trees case, each locator is compose by a pair (T, ℓ), where T is the tree identifier and ℓ is the TZ-label of a node on the tree T. In the case of a routing scheme on two levels, like the one that we are analyzing, the locator of a destination node may contain two values: the locator (T_{AS}, ℓ_{AS}) related to the destination *AS* on an AS-level tree T_{AS}, plus the locator (T_r, ℓ_r) of the destination router r on an inter-AS tree T_r.

When a packet has a destination locator like $(T_{AS}, \ell_{AS})/(T_r, \ell_r)$, the network first tries to forward the packet toward the locator on the top of the stack. Once the packet reaches the destination (T_{AS}, ℓ_{AS}), the first router pops the address from the stack and start to use the second locator, which is (T_r, ℓ_r).

It may happen that, at some point in the forwarding process, some router is not able to route a packet using the address at the top of the destination stack, because it does not have enough knowledge. This is the case when a local router needs to forward a packet on a global tree. In this case the local router pushes a new locator (T_g, ℓ_g) on top of the stack, where T_g is a local tree known by the local router, and ℓ_g is the label of a gateway router, that is able to route the packet on the global trees. When the packet reaches the gateway ℓ_g, the gateway pops the first label and forward the packet on the global tree.

As in the case of content-based forwarding it may happen that a packet that reaches a new AS needs to cross it to reach another one. Also in this case, the gateway router forwards the packet on a local tree to the right gateway using TZ-labels.

Figure 3.9 shows an example of packet forwarding in our hierarchical routing scheme, where we combine the usage of content-based and locator-based forwarding to send a request packet and its related reply packet. In the picture the two big gray ovals represent two ASes, called $AS1$, the one at the bottom, and $AS2$, the one at the top. The two ASes are spanned at the global level by two trees, one represented in red and one in green. Internally each AS is covered with a single tree, represented with black edges. Inside each AS we have internal nodes and gateway routers, represented as described in the key at the bottom of the picture.

In this example, node A, in $AS1$, sends a request packet with a descriptor D that matches the predicate announced by node B, which is in $AS2$. The initial request packet is represented in the the Figure 3.9a. The request packet contains the descriptor D and the locator of node A, which is (T_{AS1}, TZ_A), in the source stack. Node A is a local router and does not have the knowledge of the predicates announced by other ASes, therefore A does not know how to forward the request packet. For this reason node A needs to send the packet to a gateway router, which is node $G2$ in the example. To forward the packet, A pushes the locator $(T_{AS1}, G2)$ of $G2$ in the destination stack of the request packet. The network forwards the packet to $G2$ using the locator-based forwarding, indicted with yellow arrows.

Figure 3.9b shows what happen when the request packet reaches the gateway router $G2$. First of all, $G2$ pops its own label from the destination stack, because the request reached the final destination. Then $G2$ performs a content lookup on its FIB and realizes that the request needs to be sent to the autonomous system $AS2$. $G2$ selects a global tree, pushes the locator of $AS1$ for the select tree in the source stack of the request packet, and forward the packet to $G1$ in $AS2$. In Figure 3.9b $G2$ selects the green tree as a global tree, so the source label of $AS1$ to push in the source stack of the packet is $(T_{green}, AS1)$.

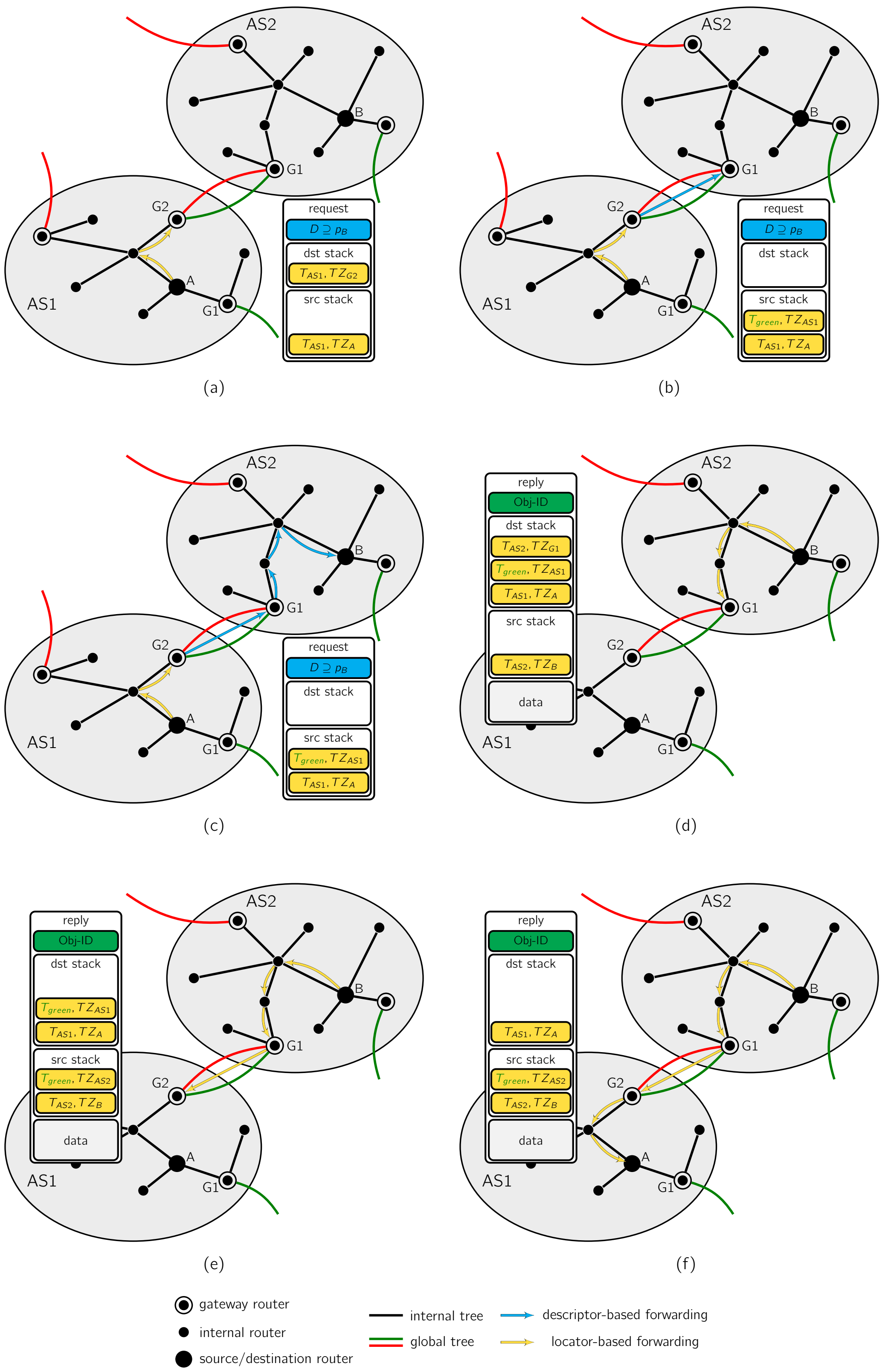

Figure 3.9. Packets forwarding on hierarchical multi-trees

When the request packet reaches node $G1$ at $AS2$ (Figure 3.9c), node $G1$ simply matches the descriptor D on its FIB and forwards the request toward router B.

Router B is the publisher node, and it has the data requested by node A. In Figure 3.9d we show the data packet created by the router B. The data packet contains a content identifier, indicated with the green box labeled Obj-ID, and, in the source stack, the data packet has the locator (T_{AS2}, TZ_B), which is the locator of node B. In the destination stack of the data packet, node B copies the source stack of the request packet, namely $(T_{green}, TZ_{AS1})/(T_{AS1}, TZ_A)$. In addition node B pushes the locator (T_{AS2}, TZ_{G1}) of the gateway $G1$ in the destination stack, because node B is a local router, and it does not know how to route on a global tree. The forwarding of the data packet from node B to the gateway $G1$ is done using locator-based forwarding.

In Figure 3.9e the data packet arrives to node $G1$. $G1$ removes its own locator from the destination stack, forwards the packet on the global tree T_{green} according to the locator at the top of the destination stack, and adds the locator (T_{green}, TZ_{AS2}) of $AS2$ in the source stack.

In the last step (Figure 3.9f) node $G2$ removes the locator (T_{green}, TZ_{AS1}) from the top of the destination stack, because the request is now inside AS_1, and forwards the packet to node A using the locator (T_{AS1}, TZ_A). Node A, at this point, has the required data. In addition node A knows the locator of node B in $AS2$, therefore A can send follow up requests using the locator of B through locator-based forwarding.

3.5 Routing Information Based : Representation and Maintenance

In this section we describe a concrete implementation of the Routing Information Base (RIB) for our routing scheme. On each router v we need to keep some state for every tree T. In particular each router needs to store:

- the adjacency list adj_v^T of v on T, meaning the subset of v's neighbors adjacent to v also on the tree T.
- the TZ-label ℓ_v^T of router v on T.
- the map $P_v^T : w \rightarrow P_{T,w}$ that associates each neighbor w in adj_v^T with a predicate $P_{T,w}$, where $P_{T,w}$ is the set of content descriptors announced by the nodes reachable through w.

The goal of our implementation is to create a data structure that is compact and easy to maintain, to allow scalability both in terms of space and update time. In the list of information to store on the routers the only complex part is the map P_v^T. In fact adj_v^T and ℓ_v^T require minimal space and they do not need frequent updates, because they are quite stable, like the trees used in our routing scheme, therefore we can use a simple data structure to store this data (e.g a vector or an hash table). On the other hand, the map P_v^T changes with the changing of application preferences (content descriptors) and is also by far the heaviest component of the RIB. In this section we focus on the implementation of P_v^T and we show how we can compress and update this data structure.

3.5.1 RIB Minimization: Compression Techniques

The amount of the information that we need to store in the RIBs of our routers can be quite large. In case of gateway routers we need to store all the predicates announced by all the ASes. For this reason the compression of the RIBs is fundamental. In this section we present three compressions that we apply to our data in order to minimize the size of the routing table on each node.

The first compression comes from the *subset aggregation*. As described in Section 2.2.4, descriptors aggregate by subset relation. If a router has two descriptors D_1^i and D_2^i associated to the same interface i, and $D_1^i \subseteq D_2^i$, then the router can discard the descriptor D_2^i and store only D_1^i, which is the more general one. In fact, every packet that matches D_2^i also matches D_1^i.

The second compression is due to the *bloom filter* representation of the descriptors, introduced in Section 3.2. This representation avoids the storage of sets of strings and gives us a compact way to represents set of tags. Thanks to bloom filters, we represent all the descriptors with a fix length in the packet header, and this is useful when we need to process the packet for the matching procedures.

The last compression that we apply to the RIBs is described in Figure 3.10. On the left side of the figure there is a RIB generated by two trees (a red and a green one) implemented as a concatenation of two RIBs, one for each tree. In this representation there are many redundant data. Each descriptor, represented as a sequence of 8 bits, appears multiple times in the table, associated with different interfaces. In the picture each descriptor is also associated with a letter that helps to spot duplicated descriptors: if the same letter appears more than once the descriptor is replicated. Descriptors replication is mainly due to two factors. Popular descriptors have more chances to appear multiple times in the same table, just because many users or applications use them. The second reason

RIB	
tree T, next-hop w	predicate $P_{T,w}$
T_{green}, c	00100101 (a)
	01010000 (b)
	01000001 (c)
T_{green}, f	00100100 (d)
	01010000 (b)
	10011000 (e)
T_{green}, e	00010000 (f)
	10000101 (g)
T_{red}, c	00100101 (a)
	01000001 (c)
T_{red}, e	00010000 (f)
	10000101 (g)
	00100100 (d)

Compressed RIB	
descriptor D	tree T, next-hop w
(a) 00100101	(T_{green}, c), (T_{red}, c)
(b) 01010000	(T_{green}, c), (T_{green}, f)
(c) 01000001	(T_{green}, c), (T_{red}, c)
(d) 00100100	(T_{green}, f), (T_{red}, e)
(e) 10011000	(T_{green}, f)
(f) 00010000	(T_{green}, e), (T_{red}, e)
(g) 10000101	(T_{green}, e), (T_{red}, e)

Figure 3.10. RIB index by descriptors

for duplicated descriptors is the usage of multiple trees. Since we replicate all the routing state for each tree there is an high probability that a router sees the same descriptor over different trees.

Descriptors are the most expensive component to store in the RIB: in our implementation a descriptor requires 24 bytes, while a tree-interface pair can be stored in only 2 bytes. For this reason, we avoid to store descriptors multiple times, and we use the representation on the right side of Figure 3.10, where we *index the RIB by descriptors*. In this representation the RIB contains only unique descriptors, and, for each descriptor, there is a list of tree-interface pairs, used to find the next hop in case of a matching packet.

3.5.2 RIB Representation: Data Structure

The representation on the right side of Figure 3.10 is the most tight way to represent our RIB. Unfortunately it is not possible to use this data structure in practice, because we need to update the RIB. A table like the one in the picture requires a linear scan for each descriptor that we want to add or remove from the table. A linear scan can be really costly in case of large RIBs. In our implementation we need a data structure that is compact as the table representation, but also easy to modify.

When we process update packets for our routing protocol, we want always to keep the RIB minimal. This means that, for each descriptor D_1^i associated with interface i, there must be no other descriptor D_2^i on the same interface i such that $D_2^i \subseteq D_1^i$. To guarantee this property, whenever a descriptor D needs to be added to the RIB, the router needs to search for subsets or supersets of D in the table. Similar checks are required also in the case that a descriptor needs to be removed from the RIB. We need a data structure that allows subset and superset check operations, which are the building blocks of our update algorithm, in a reasonable amount of time.

To summarize, we need a data structure that requires small amount of memory and allows fast implementation of the subset and superset check. In our implementation we use a PATRICIA trie [56] to represent our RIB. An example of PATRICIA trie that stores the descriptors of table in Figure 3.10 is presented in Figure 3.11.

The PATRICIA trie requires minimal space because, as shown in the figure, each descriptor appears only once. The data structure requires two pointers per nodes, but thanks to these pointers we can simply rearrange the data structure to store new descriptors. Each node also has an extra pointer to keep track of the next-hop information, represented as a list of 2 byte tree-interface pairs.

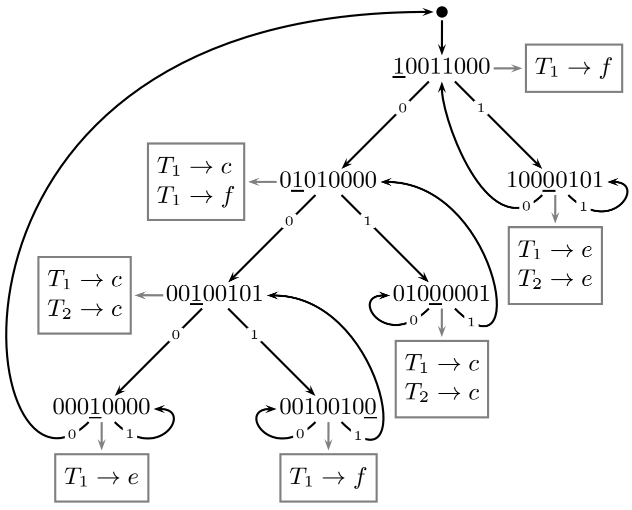

Figure 3.11. PATRICIA trie used to store the RIB

The PATRICIA trie admits to a straightforward implementation of the subset and superset check, which amount to simple walks on the trie. Thanks to the PATRICIA trie structure we can skip some branches of the data structure during the subset checks: if we look for subsets of a descriptor D, and D contains a zero at a position identified by a node n, we can skip the entire subtree under n that contains a one in that position. A similar strategy works also in case we look for supersets of D.

In the implementation we also divided the trie in multiple tries, one for each Hamming weight (number of the 1s in a bloom filter). This allows us to skip entire tries with descriptors that contains too many (or too few) ones with respect to the descriptor that we are trying to add (or remove). In addition, this allows parallelizations, since each trie is independent from the other. In our implementation we use multiple threads to process multiple tries simultaneously.

3.5.3 RIB Maintenance: Update Algorithm

Routing information propagates in the network through update packets, each representing an incremental change. Thus an update packet contains a set of descriptors to be added and a set of descriptors to be removed. We refer to these sets as a routing *delta*.

Each router processes an update packet using the algorithm presented in Figure 3.12. The algorithm consists of a main function called *apply_delta* (line 1) that processes a packet *update* received from interface *ifx* on the tree T. The *apply_delta* function calls *remove_descriptors* (line 4) for all the descriptors in the removal delta, and then *add_descriptors* (line 6) for all the descriptors in the addition delta.

The processing of a routing update may produce changes in the local RIB, and may also trigger new updates to be forwarded to neighbor routers. The *apply_delta* function returns this set of updates through the *result* map, which is an input/output parameter passed to the *apply_delta* function.

In function *add_descriptor* (line 8) we try to add a new descriptor to the RIB. First, we check if the new descriptor d is not a superset of other descriptors associated with the same interface and tree from where the router received the update. If no subset descriptor exists, we add d, and we remove all the supersets of d from the incoming interface *ifx* on tree T. This procedure guarantees that the RIB will always be minimal (no redundant descriptors). The router must then propagate descriptor d as an addition delta for all the interfaces on tree T except the incoming interface *ifx* and any other interface i that, for descriptor d, is already covered by another more general descriptor d'. We say that an interface i is covered for descriptor d by a descriptor d' when there exists another interface i' on T associated with a certain descriptor $d' \subseteq d$. In practice, this means that the router has already advertised a more general descriptor d' due to a previous update received from another interface i'. Yet in other words, the router does not need to propagate the new descriptor d because a previously added descriptor d' already covers and therefore masks d.

To clarify this point we show an example in Figure 3.13 where we show a

```
void apply_delta (map<int,delta> & result,
                  delta update, int ifx, int T) {
  for (descriptor d : update.removals)
    remove_descriptor(result, d, ifx, T);
  for (descriptor d : update.additions)
    add_descriptor(result, d, ifx, T); }

void add_descriptor (map<int,delta> & result,
                  descriptor d, int ifx, int T) {
  if (!exists_subset_of(d, ifx, T)) {
    add(d, ifx, T);
    remove_supersets_of(d, ifx, T);
    for (int i : interfaces[T])
      if (i != ifx && no_subsets_on_other_ifx(d, i, T))
        result[i].additions.add(d); } }

void remove_descriptor (map<int,delta> & result,
                  descriptor d, int ifx, int T) {
  if (exists_descriptor(d, ifx, T)) {
    remove(d, ifx, T);
    for (i : interfaces[T]) {
      if (i != ifx && no_subsets_on_other_ifx(d, i, T)) {
        result[i].removals.add(d);
        result[i].additions.add(supersets_of(d, T)); } } } }
```

Figure 3.12. Incremental Update Algorithm

sequence of 4 updates received by a router. The figures come in pairs labeled (a.1) and (a.2), (b.1) and (b.2), etc., where the first and second picture in a pair describe the state of the router before and after receiving the update, respectively.

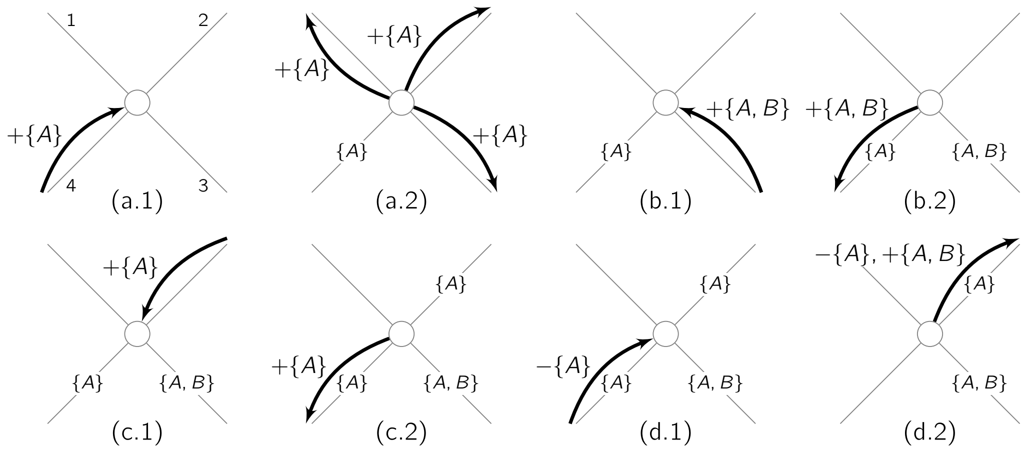

Figure 3.13. A Sequence of Incremental Updates

The router interfaces are numbered from 1 to 4 as shown in Figure 3.13a.1. We omit this numbering from the other figures for clarity. In Figure 3.13a.1 the router receives an update $+\{A\}$ from interface 4, where we indicate addition and removal deltas with '+' and '−' signs, respectively. The router adds descriptor A on interface 4, as denoted by the A edge label in Figure 3.13a.2, and forwards the update to all the other interfaces, again as shown in Figure 3.13a.2.

Then in Figure 3.13b.1 the router receives the update $+\{A, B\}$ from interface 3. Figure 3.13b.2 shows that the router adds $+\{A, B\}$ to interface 3, and then forwards the update only to interface 4. This is because both interfaces 1 and 2 are already covered by $\{A\}$, which is a subset of $\{A, B\}$, on interface 4. The same happens with the following update shown in figures 3.13c.1 and 3.13c.2, where the router sends the update only to interface 4, because the descriptor $\{A\}$ on interface 4 already covers interfaces 1 and 3.

This check is important to minimize the number of updates that a router forwards and that neighbor routers must process. For example, in the case of Figure 3.13b.2 it would be useless to send a $+\{A, B\}$ update to interface 2 because the neighbor router already has a descriptor $\{A\}$ associated with that interface due to the previous update shown in Figure 3.13a.2, so the superset descriptor $\{A, B\}$ would be discarded by the neighbor router. With this check we can decide locally which updates are indeed new for a neighbor router, thereby minimizing traffic and maintenance costs.

The removal of a descriptor is done in a similar way to the insertion. The procedure is presented at line 10 of the algorithm in Figure 3.12. However, since new additions of more specific descriptors might be masked by previous additions, removals of more generic descriptors might uncover previously masked additions. Therefore, the removal of a descriptor d may cause the addition of supersets of d, as depicted in Figure 3.13d.2. The update $-\{A\}$ is not forwarded to interfaces 1 and 3, because those are covered by descriptor $\{A\}$ on interface 2. However the update, which consists of a removal delta $-\{A\}$, is forwarded to interface 2 along with an addition delta $+\{A,B\}$. This is because the removal of $\{A\}$ from interface 4 uncovers all its supersets on the other interfaces, in particular $\{A,B\}$ on interface 3.

3.6 End-Nodes Mobility

The internet traffic generated by mobile phones increases every year, therefore address end-nodes mobility in an efficient way is getting more and more important. One of the promises of ICN is to handle end-nodes mobility in a more efficient way with respect to an IP based network. This is true in the case where a consumer node moves in the network. In CCN, for example, when a consumer moves to a new location, it simply starts to send the interests from the new position. Consumers do not generate routing state in the network, so there is no need to update the RIBs of the routers. The problem comes when we need to move a producer node. Move content means update the routing state of the network, and this process is generally costly [34].

TagNet, thanks to its naming scheme formed by multiple addresses, may solve the mobility problems in CCN. In the following we sketch an algorithm that can be used to deal with user mobility in our network architecture.

3.6.1 Consumer Mobility

The easy case to handle, as in CCN, is the mobility of consumers nodes, because they do not set any state in the network. The case of the consumer mobility is presented in Figure 3.14.

In the figure we represent a local network (the gray ellipse), spanned by a black tree. In this network there are two wireless routers that work as access points, indicated with *AP1* and *AP2*. Each one of these wireless routers has its

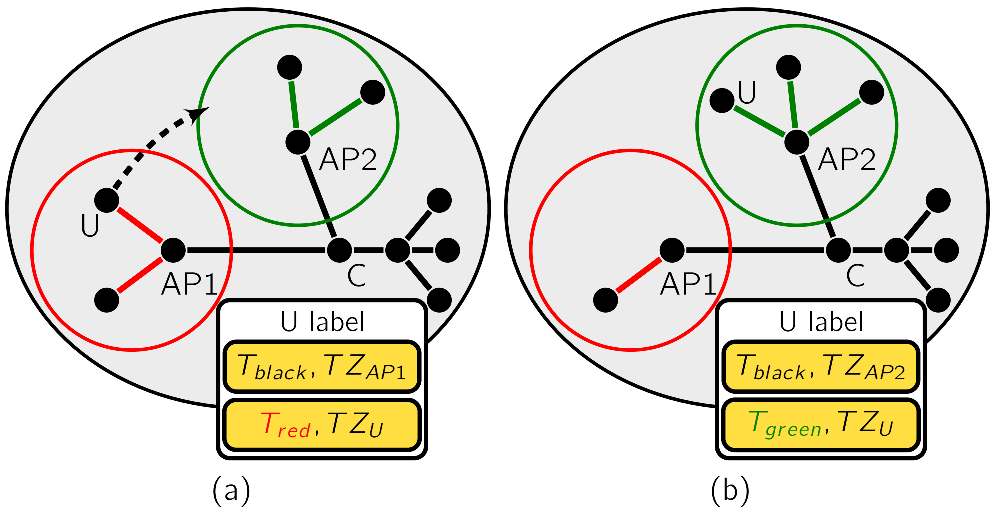

Figure 3.14. Consumer node mobility: TZ-label update

own local tree. The tree of $AP1$ is indicated in red, while the one associate with *AP2* is green. This form a hierarchy of trees at local level, that works in the same way we saw in Section 3.4. Since the routers *AP1* and *AP2* uses wireless connections, the topology formed by these routers and the end-nodes connected to them is a star topology. In this particular case we can consider the TZ-label simply as an identifier of a node. The access router can easily assign a new label to a node generating a random value, that is not already in use. Since the router does not need to run any algorithm to label all the end-nodes, this process is really fast.

The locator of node U, in Figure 3.14a, is $(T_{black}, TZ_{AP1})/(T_{red}, TZ_U)$. At a certain point node U moves from *AP1* to *AP2*, as indicated by the dashed black arrow in Figure 3.14a. As soon as U establishes a connection with *AP2*, *AP2* assigns a new label to U. The new locator of U is indicated in Figure 3.14b. In order to receive the content from its new position, node U has only to use its new locator $(T_{black}, TZ_{AP2})/(T_{green}, TZ_U)$ in the header of future request packets.

3.6.2 Producer and Subscriber Mobility

So far we described the simple case in which a consumer node moves. This case is easy to handled, since a consumer node has no state in the FIBs. Instead, in the case where a producer or a subscriber node moves, the protocol is more complex. First of all, the routing protocol will at some point update the FIBs, but that process may require a long time. Therefore, we do not rely directly on the routing algorithm in case of mobility. However, thanks to our addressing scheme, we can temporarily adapt a route to the new network configuration so that a mobile node can still provide its content or receive notifications of interest. Here we describe only the scenario where a *producer* node moves, but the case where a subscriber moves can be handled exactly in the same way.

We use once again Figure 3.14 as an example, although in this case we assume that node U is a producer node responsible for serving some content to other users. Thus in the example of Figure 3.14 node U moves from its initial access router *AP1* to another router *AP2*. The important property of our addressing scheme that allows us to handle mobility in this case is that U knows the locator (T_{black}, TZ_{AP1}) of its initial access router *AP1*, because that corresponds to the prefix of its own locator up to the very last label.

Using the locator of the initial access point *AP1*, node U transmits its new locator $(T_{black}, TZ_{AP2})/(T_{green}, TZ_U)$ to *AP1*, so that *AP1* can record a forwarding address for U in its FIB. In particular, *AP1* keeps track of all the descriptors advertised originally by U, but instead of mapping those to a local interface, it uses the new, remote locator of node U. Therefore, when *AP1* receives a request packet with a descriptor that matches the content announced by U, *AP1* encapsulates and resends the request in a new packet addressed to U's new locator. The same happens when *AP1* receives a packet that has as a destination label the initial locator of U. Notice that this indirection applies only to the first packet (or the first few packets) in the communication flow, since U will reply to the first request with its new locator, thereby instructing the consumer to send every subsequent requests to its new location.

The process that we just described is useful to temporarily re-route the packets in the network, without waiting for the routing protocol to update the FIBs of all the routers. This can be useful, for example, in case of VoIP applications, to keep the communication persistent. With this simple temporary indirection is possible to bypass the routing update process. However update the routing state is important and it must be done timely. The update process is described in Figure 3.15.

In Figure 3.15a, node U sends an update packet, indicated with orange arrows. The arrows are labeled with a "+", since the update packet contains the descriptors that need to be added in the RIBs of the routers, in order to steer relevant request packets toward the new position of U. Although the update should be broadcast through the entire network, the update is forwarded only on the path that connects the actual access router of U, namely *AP2*, with the old access router of U, namely *AP1*. In fact, as described in Figure 3.15a, the update

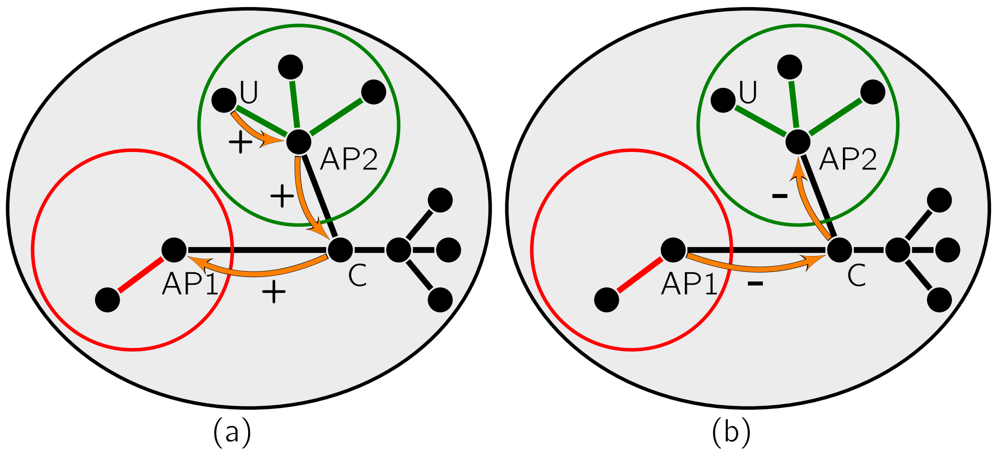

Figure 3.15. Publisher/Subscriber node mobility: RIBs update

process does not involve the nodes on the right of node C. This happen because the old state generated by node U at its previous position in *AP1*, prevent C from forwarding the update farther, as described in the algorithm in Figure 3.12. This is an important feature of the update algorithm that keeps the changes as local as possible.

Now that the routing state is updated, node U can be reached directly. However we still need to remove the old state that points to *AP1* from the RIBs. When *AP1* receives the update from U, *AP1* sends the same update through the network, this time indicating that each router needs to delete the descriptors in the update packet, as indicated by the sign "−" in Figure 3.15b. Notice that also in this case the update process involves only the nodes on the path between node *AP1* and *AP2*.

What we described in this section is a way to handle end-nodes mobility in TagNet. Thanks to the hierarchical routing protocol and the naming scheme that supports multiple addresses, we can easily forward packets to mobile nodes. In TagNet we can immediately create a valuable route to the new position of a publisher node, using locators. At the same time the routing protocol is able to avoid updates flooding, because our update algorithm minimizes the amount of update messages that we need to send.

3.7 Evaluation

In this section we present the evaluation of our routing scheme. There are three main questions that we need to address in order to prove the feasibility of our proposal:

1. Is it possible to cover the Internet with trees in an effective way?
2. How big is the routing table of a core router?
3. Is it possible to update big routing tables in a reasonable amount of time?

As state previously in Section 3.3.4, the main focus of this thesis is the scalability aspect of the routing scheme. For this reason we present only preliminary results for the first question, while a more deep and extensive study is part of a another thesis. Regarding the research questions 2 and 3, here we present an extensive analysis of the scalability of our routing scheme, both in term of space required on each router and in terms of update time of the RIBs.

A crucial difficulty in conducting this analysis is that there is no known deployment of an information-centric network at the scale we are targeting. Therefore, we must use synthetic workloads. Below, we explain how we create such workloads, in particular we explain how we generate descriptors for each user, and how we spread them over the ASes.

3.7.1 Tag-Based Descriptors Workload

The objective of this analysis is to create a workload that can correspond to a plausible behavior of the users of an information-centric network. In particular, for our experiments, we are interested only in that set of information that contribute to create the routing state. We create descriptors used by publisher nodes to announce content that can be retrieved by consumers, and descriptors used by subscribers to specify their interests. We analyze four sets of applications: a push active web service, that sends related post and content to interested users, a pull on-demand video content retrieval service, a push micro blogging service and a pull torrent application used to download files.

In the following of this section we describe how we collect data for all the applications, how we amplify them in order to get a bigger workload, and how we spread users over the ASes.

Active Web

We envision a future information-centric network used to actively distribute Web content. We analyze the interests of users, in order to identify their subscriptions. This is the information that we need to store in the RIBs for the active Web.

To derive a set of plausible subscriptions we analyze users bookmarks. In particular we use the bookmarks collection of the Delicious website,[2] which contains the public bookmarks of about 950,000 users retrieved between December 2007 and April 2008 [83]. The data set contains about 132 million bookmarks and 420 million tags. We assume that users are interested in the content they bookmark, and we create subscriptions with the tags they assign to their bookmarks. Therefore, we derive plausible subscriptions from user tag sets. In total we derive 23,248,896 subscriptions for 922,651users.

We also analyze data collected from blogs. In particular, we study the Blog06 collection from the Text Retrieval Conference (TREC),[3] which contains 3,215,171 blog posts from 100,649 unique blogs. We apply the Latent Dirichlet Allocation (LDA) algorithm [6] to extract and categorize the posts under a set of topics. We then assume that an user has an interest in a specific topic if the user writes at least two relevant posts on that topic. For each topic, we select the 10 most relevant tags and we use them as a descriptor for the subscriptions of a user. With this analysis we identify 59,185 blogs with 178,189 posts from which we derive subscriptions.

On-Demand Video Content

In a future information-centric network users will be able to download or watch videos on-demand, in a similar way to what we do nowadays with YouTube. We analyze a collection of data related to YouTube in order to figure out how users publish their content, which in our network determines the descriptors of the content offered by the publishers. In particular we look at the tags that publishers assign to their videos to allow viewers to find those videos with keywords based search. These keywords were publicly visible until few years ago. Since now the tags are not available anymore, we analyze a data set derived from 10,351 videos published by 782 users in the "Politics" category. Unfortunately this data set is not available anymore.

[2]Delicious website. https://delicious.com/

[3]The Blog06 test collection. http://ir.dcs.gla.ac.uk/test_collections/blog06info.html

Publish/Subscribe Micro Blogging

For this application we analyze Twitter. In particular we envision a Twitter based on a publish/subscribe service. A set of subscriptions of a user contains the list of followed accounts plus a set of interests, specified with hashtags. Messages published by a certain user are forwarded to all the followers of such users, plus all the users interested in the hashtags in the message. To derive our subscriptions we use a graph of 41.7 million Twitter users and 1.47 billion follower relations [53], while for the hashtags we use a collection of 16 million tweets recorded during two weeks in 2011. This dataset of tweets was provided again by the TREC conference (2011-2012).[4] In order to derive subscriptions from the messages, we considered only the messages with both hashtags and URLs. We say that a user is interested in the content of a certain URL when the user publishes the URL in a tweet. We use the set of hashtags in the message to create the subscription for such a user. In total we collect 446,370 subscriptions for 349,753 users

Torrent Application

We analyze also a set of data collected from a BitTorrent client. We use the set of tags used to search specific content as the descriptors for the content published by a user. We use a dataset of 9,669,035 queries collected over a period of 3 months from kickasstorrents.com by the Computer Networking Research Laboratory of Colorado State University.[5] This dataset contains 1,353,662 unique tags.

Data Amplification

The extrapolated workloads suffer from two limitations: they are too small for the kind of experiments that we want to conduct, and they are biased, due to the fact that almost all the data that we have are in English.

In order to amplify the workload and remove the bias toward the English language we expand the workload considering the most spoken 25 languages in the world. When we create a new user in our workload we select a language, according to the distribution of the number of native speakers of those 25 languages, and we generate a set of descriptors in that language. We do

[4]The Tweets2011 collection. https://github.com/lintool/twitter-tools/wiki/Tweets2011-Collection

[5]BitTorrent search terms dataset. http://www.cnrl.colostate.edu/Projects/CP2P/BTData/

not want to lose the semantic correlations between tags in the datasets, therefore we create artificial translations of the tags, adding a prefix to each tag that indicates the corresponding language. For example, if the English data set contains the tag “movie”, the Japanese “translation” of such tag is indicated with the tag “Japanese_movie”. The selection of a language is independent from the geographical location of the user. This is not completely realistic, but it creates a workload that gives us less chances to aggregate descriptors, so it represents a kind of worst case scenario for our routing protocol.

To further expand the workload and also to avoid creating exact replicas of the original dataset, we create additional tags using synonyms. Again synonyms are created artificially. As in the previous example, the tag “Japanese_movie” can have as a synonym the tag “Syn1_Japanese_movie” or ‘Syn2_Japanese_movie”. We assume that each tag has 2 synonyms and we select one of the possible three tags with uniform probability.

Users Distribution

Now that we have our workload we need a way to realistically disseminate users on different ASes of the Internet. In order to assign to each AS a plausible number of users we use the number of IP addresses announced by each AS and the number of users that have access to the Internet in the countries covered by the AS.

We collect the set of IP address announced by each AS in our topology. We use the whois service to get all the data.[6] Using the GeoLite database,[7] we assign a country to each IP address. With this data we discover the countries covered by each AS and the percentage of IP addresses owned by each AS for each country. For example, according to our statistics Orange (AS3215) owns almost 25% of the IP addresses in France, so we assign to Orange 25% of the French Internet users. We collected the statistics for the Internet users of each country of the world on the Internet Word Statistics website.[8] In the end the total number of users associated with an AS is the sum of all the users that we assign to the AS for each country covered by the AS.

[6]we use the service provided by https://www.shadowserver.org/wiki/. With the command whois -h asn.shadowserver.org “prefix ASN” (where ASN is the number that identify the AS) it is possible to retrieve all the IP prefixes announced by an AS.

[7]GeoLite2 free downloadable databases. http://dev.maxmind.com/geoip/geoip2/geolite2/

[8]Internet Word Stats website. http://www.internetworldstats.com/stats.htm

3.7.2 Internet Topology and Trees

In this section we show the results of an experiment the we conduct to demonstrate that is possible to effectively cover the Internet topology using few trees. Here we show only data related to the additional cost of using few trees, but we also conducted other experiments where we measure other parameters [60, 61].

In this experiment we want to measure the traffic overhead incurred by our tree-based routing scheme. We conduct our analysis on the Internet AS-level topology consisting of a graph of 42113 nodes and 118040 edges.[9]

We construct sets of k trees over the topology as follows: we choose k nodes at random by selecting Tier-1, Large ISP, Small ISP, and Stub ASes with probability 40%, 30%, 20%, and 10%, respectively. In particular, Tier-1 are ASes with no providers, Large ISPs are ASes that serve a customer tree of 50 or more ASes, Small ISPs have between 5 and 50 customers and Stubs less than 5. We then use k shortest-paths trees, each one rooted at one of the k nodes.

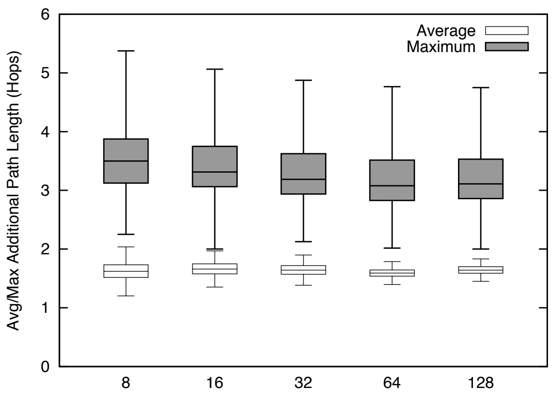

Figure 3.16. Maximum and average additional cost in forward packet over trees

For each value of $k = 8, 16, 32, 64, 128$ we analyze for all pairs of ASes (nodes of our graph), the average and maximal additional path lengths over the k trees, which gives an indication of the traffic overhead. The data in Figure 3.16 are the aggregated values over 20 runs of our simulation. We changed the root ASes at every run.

For each value of k trees, the top and bottom box plots show the distributions of the maximum and average (i.e., expected) additional path lengths, respectively. The box plot cover the distribution from the 1-st to the 99-th percentile, showing also the 25-th, 50-th and 75-th percentile. The results show

[9]Internet AS-level topology archive (data retrieved 29/06/2012). http://irl.cs.ucla.edu/topology/

that k-trees can be used to approximate an optimal path on the Internet with a minimal additional cost. In expectation, we have an additional cost of around 1.5 hops, no matters how many trees do we use. In addition, using only 8 trees the maximum additional cost is less than 6 hops. Notice that, with more trees, the value of the maximum additional cost does not change much, and this result suggests to use only few trees to cover the Internet network.

3.7.3 Scalability: Memory Requirement and Maintenance

Now we evaluate the memory requirements of our ICN routing scheme. We generate a workload for 50 million users using the workload generator described in Section 3.7.1.

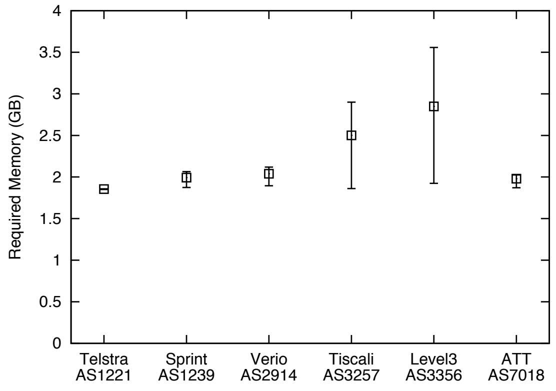

Figure 3.17. RIB sizes on gateway nodes on different ASes

In Figure 3.17 we show the memory required by the RIBs of gateways routers on different ASes. As described in Figure 3.8 in Section 3.4, each gateway router may participate only in a subset of the trees used to cover the AS-level network. To know exactly how many trees each router needs to store we should know the exact connectivity among AS at their router level, but this information is not publicly available. We decide then to consider all the possible cases and see what would be the memory required on a router using all possible combination of trees. The plot shows the minimum, the maximum and the average amount of memory that would be needed to store the routing information from 1 up to 8 trees.

The variation that we see among different ASes is due to the different degree and location of the ASes in the network. Usually an AS with many neighbors experiences less compression, because the descriptors are spreaded over many interfaces. Notice, however, that the absolute values are relatively low: the most demanding case, which correspond to Level3, requires less than 3.6GB of memory. In this evaluation section we always measure the total size of the data structure presented in Section 3.5.2, not only the amount of memory required to store the descriptors. Another interesting result is that the memory required by 8 trees, indicated by the maximum value, is always less than twice the memory required by a single tree (the minimum value). This means that, in our scheme, descriptors aggregate well across trees, thanks to the compression presented in Figure 3.10.

For each AS we also measure the memory required to store the RIBs on routers at the intra-AS level. In this experiment we use the internal AS topologies available from the Rocketfuel project [74]. The data of this evaluation are presented in Figure 3.18. The N and E labels in the graph represent the number of nodes and edges in the topology of each AS, respectively. For the intra-AS analysis we create one shortest path tree for each node in the topology, therefore, for example, for Level3 we generate 624 trees. In this way the latency and the edge load on the trees is equal to the one on the total graph.

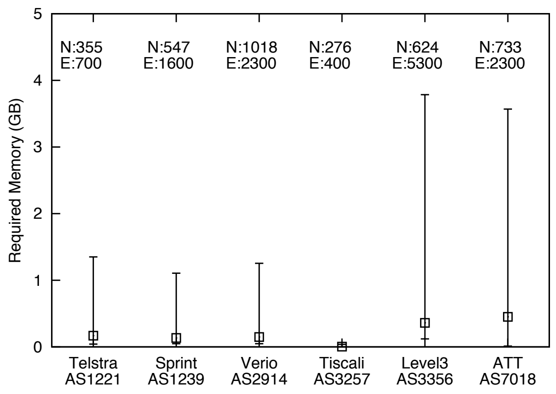

Figure 3.18. RIB sizes of nodes inside different ASes

The plot represents again the minimum, the maximum and the average sizes of the RIBs used to store local trees. In particular, the distribution in the graph refers to the sizes of 50 nodes sampled randomly from the networks. Considering the worst case results, namely Level3 and AT&T, we can see that, even using hundreds of trees to cover the network, we obtain good levels of aggregation and good results in absolute terms, with a maximum memory requirement of less than 4GB.

So far we presented results for a workload of 50 million users. This is a relatively low number with respect to the current Internet users population. In order to better demonstrate the scalability of our routing scheme, we focus on AS 3257 and on a single shortest-paths tree to study the memory requirement under a workload of almost 10 billion content descriptors, corresponding to 500 million users. Figure 3.19 shows the memory required at a gateway router for an increasingly larger user population.

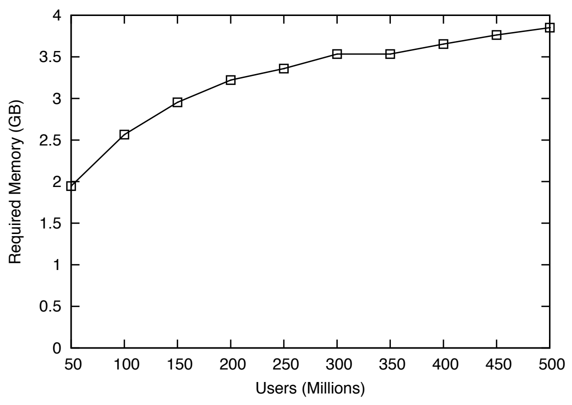

Figure 3.19. RIB size scalability for different workload sizes

We can see that the growth of the memory requirement is relatively high initially, but steadily flattens, reaching 3.8GB for 500 million users. This shows that the subset aggregation used in our routing scheme is indeed effective. Increasing the number of the users, the memory required to store all the descriptors remains almost constant, since most of the new descriptors will be aggregated to old ones at no additional cost. If we consider the growth rate between 450 and 500 million users to be constant, which amounts to 0.09 GB for 50 million users, we need around 8.3 GB to store the descriptors for 3 billion users, which is approximately the actual Internet population.

In the last experiment we evaluate the time required by our update algorithm, described in Section 3.5.3, to add or remove a descriptor from the RIB. In this analysis we start from the RIBs computed for the experiment in Figure 3.19 and then apply updates with a total number of 20, 40, and 60 descriptors. In each update the number of descriptors to add or remove from the RIB is randomly chosen. We run our algorithm on a commodity PC, that has two Intel Xeon E5630 processors, with four cores each, and 2.53HHz clock frequency. The system is equipped with 16GB of RAM.

The plot in Figure 3.20 shows the results of this experiment, in particular the median and standard deviation of the update time computed over 1000 updates. The data points on the three lines are for the same values of the x-axis, but for purposes of readability have been slightly shifted to avoid obscuring each other.

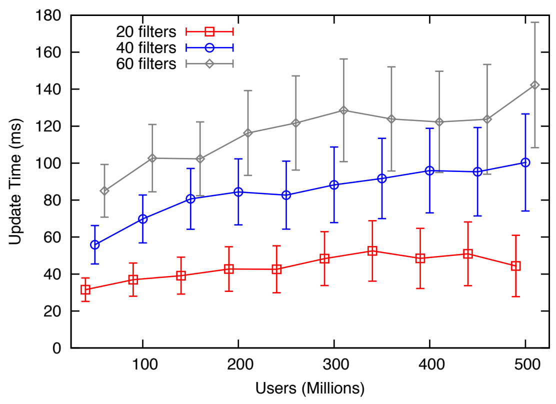

Figure 3.20. Scalability of the maintenance time

The most important result is that the update time does not increase significantly with the size of the RIBs and is almost constant for each descriptor. To process one descriptor, we need on average 2.04ms, so our algorithm can handle about 500 updates per second on large RIBs. According to the BGP Instability Report, the average number of BGP update messages per seconds is 4.62, which means that our router can easily handle the normal traffic generated by BGP on the Internet.[10] However the same report also highlights a peak of 5708 BGP update messages per second, which is 10 times more than what is sustainable by our router.

[10]The BGP Instability Report. 7 Day BGP Profile: 9 November 2015 00:00 - 15 November 2015 23:59 (UTC+1000) http://bgpupdates.potaroo.net/instability/bgpupd.html

Chapter 4

Matching and Forwarding Algorithm

TagNet provides three kinds of packets: notification, request and reply packets. Each packet has its own semantics and is forwarded in a specific manner, although using a single data structure. Notification packets are forwarded using the descriptor in the packet header that describes the content of the notification itself. A router forwards a notification packet in multicast in order to reach all interested users. A request packet can be forwarded in two ways. The content-based forwarding uses the descriptor in the packet and sends the packet to one interface. The locator-based forwarding uses the locator of the publisher node, and this forwarding algorithm uses the TZ-labels. Finally, the reply packets are always forwarded using the TZ-label of the requester node.

In order to forward the packets using descriptors we need to address the partial matching problem, described in Section 4.1. We say that a packet with a descriptor P matches a descriptor D in the FIB if $P \supseteq D$. We present two algorithms to match packets using descriptors. The two algorithms use a single and common data structure. The first algorithm, used for the notifications, finds all the matching descriptors in the FIB, in order to send the packet to all the matching interfaces. The second algorithm, used for request packets, forwards the packet to the interface with an associated descriptor that is the largest subset of the packet descriptor available in the FIB. In this way the router forwards a request packet toward the most relevant source. This algorithm can also be used to mimic the longest prefix match semantics of hierarchical names.

For TZ-labels we implement the algorithm proposed by Thorup and Zwick for their compact routing scheme on trees [76]. This algorithm is quite simple and requires to store only few bytes of information on each node. It also achieve high throughput in the forwarding process.

In this chapter we describe and evaluate the implementation of a matching engine that is able to forward all the defined packets in the appropriate way. In this sense, our forwarding is complete. In contrast, most of the forwarding engine proposed for CCN focus their attention on the forwarding process of one specific kind of packet, in particular interest or data packets. This is due to the fact that CCN uses different data structures for each message, and so the forwarding process is more complex.

The Chapter is structured as follows: in Section 4.1 we describe in details the Partial Matching Problem. In Section 4.2 we describe some related work. Section 4.3 describes the descriptor-based matching algorithms, while Section 4.5 provides an overview of the TZ-label scheme and the matching algorithm based on locators. We then evaluate our implementation in Section 4.6, where we show that our forwarding engine can achieve 20Gbps of throughput using commodity hardware with different workloads.

4.1 Partial Matching Problem

The descriptor-based matching in TagNet corresponds to a well-known combinatorial problem called *subset query problem* or *partial matching problem*. The original problem is defined as follow: D is a dictionary of sets, all subsets of a universe set U. Given a query set $Q \subset U$, find all the matching sets $S \in D$, such that $Q \supseteq S$. In our setting, the dictionary D is the FIB of the router, and the query Q is the descriptor in the packet header.

This problem is well studied in different fields, because it has many relevant applications. In the networking area the partial matching is use to implement packet classification [42]. Incoming packets are classified according to a set of rules. Packets in different classes can be handled by the network in different ways. In information retrieval, the partial matching is used to search documents, or collection of data that contains a given set of words. This is, for example, what happens in a search engine like Google. Also databases use this kind of search. An example of database that implement partial matching is MongoDB.[1]

There are two trivial solutions to this problem. The first solution is to store all the answers for all the possible queries. When a packet comes, the matcher needs a single look-up in this index to get all the possible results. The second solution requires a linear scan of the entire FIB, in order to find all the matching descriptors. Unfortunately, none of these algorithms is usable in practice. The first implementation requires 2^m space, where m is the size of the query. In our case $m = 192$, that corresponds to the number of bits in the bloom filter

[1]MongoDB website. https://www.mongodb.org/

implementation of the descriptors. For this reason, this first trivial solution is unfeasible, because the memory requirement is to high. In the second solution we need to store only the descriptors of our FIB, but the algorithm requires $O(Nm)$ query time, where N is the number of entries in the FIB. This algorithm is again not usable for our matching engine, because N could be really high, and so the throughput of the matcher would be low.

Since is not trivial to find a good algorithm for the partial matching problem, there is a conjecture that says that this problem suffers from the course of dimensionality [7]. This means that probably there are no algorithms that achieve both low memory usage and fast query time at the same time. In this chapter we show how we tackle the problem combining algorithmic and engineering aspects, in order to reduce the size of the FIB and achieve a good query time.

4.2 Related Work

We start the related work section talking about the literature related to the partial matching problem. Then we examine some matching algorithms proposed for CCN.

The first non trivial algorithm for the partial matching problem was proposed by Rivest [69]. Rivest shows that it is possible to reduce the space complexity of the trivial solution that requires 2^m space, in the case where $m \leq 2\log N$. However, this reduction is not useful for us, since it would only apply to FIBs of over 2^{96} entries. In the same paper the author proposes an algorithm based on a trie, which is quite similar to the one that we propose. The approximate query time for this algorithm is $N^{\log(2-s/m)}$, where, in our sitting, s is the number of 0s in the query descriptor. In the same paper Rivest conjectures that the lower bound for any possible algorithm for the partial matching problem is $N^{(1-s/m)}$.

Charikar et al. propose two improvements over the trivial solutions [21]. The first algorithm requires $N2^{O(m\log^2 m\sqrt{c/\log N})}$ space and $O(N/2^c)$ query time for any constant value c, while the second requires Nm^2 space and $O(mN/c)$ time for any $c \leq N$. Although these results are interesting from a theoretical point of view they are not really useful in practice, because either they require too much memory, or they are too slow for a real implementation of a line speed matching engine.

Beyond the theoretical and more general results, there are some practical solutions for particular instances of the problem. Among them, there are algorithms that exploit specialized hardware, such as TCAMs. A TCAM is an associative (or "content-addressable") memory. Given a particular word, the memory returns one of the addresses where the word is stored. More specifically, a TCAM is a ternary content-addressable memory, meaning that each bit can be either 0, 1, or $*$ (that indicates a “don’t care”). TCAMs are already used for fast IP matching [90, 66] and they admit a direct implementation of the subset check [39]. TCAMs, however, have many limitations, in fact they are expensive, they consume a lot of power, and they are too small for our application domain. In addition a TCAM returns only one matching entry. In our setting, this can be useful to forward request packets, but it can be quite complicated to implement a matching engine for notification packets using TCAMs. In case of notification packets in fact we need to return all the matching interfaces.

In the context of CCN, there are many proposed algorithms and implementations for a named-based matcher. Wang et al. propose a matcher that uses a GPU to improve the performances [82]. This implementation covers only the matching of the interest packets. In the same year So et al. propose a complete solution for the forwarding engine of an NDN router [73]. Perino et al. describe another solution for a name-based matcher using network processors [63]. Also in this case the authors cover only the interest matching. All the implementations achieve quite high throughput, around 10 or 20 Gbps, depending on the implementation. It is worth to notice that these matchers are designed for hierarchical names. Hierarchical names require longest prefix match, that is much simpler than the partial matching problem that we have to implement. Nevertheless, our implementation achieves similar throughput, with bigger FIBs, using only commodity hardware machine. This is possible thanks to the combination of matching algorithms based on descriptors and locators.

4.3 Content-Based Matching Algorithm

In this section we focus on the content-based matching, that is the matching based on descriptors. In order to describe the matching algorithm and all the implementation details we proceed as follow: in this section we give a high-level description of the data structure and the content-based matching algorithms (presented in Section 4.3.1 and Section 4.3.2); we describe some algorithmic improvements in Section 4.3.3; in Section 4.4 we show all the implementation details and the implementation-specific optimizations.

To implement our matching algorithm we use a prefix trie, where we store all the descriptors of the FIB, with attached the forwarding information, namely the output ports related to each descriptor. Figure 4.1 shows the basic data structure.

In this trie each path from the root to a leaf represents a descriptor in the FIB. A descriptor is identified as a sequence of positions (from 1 to 10 in the picture, from 1 to 192 in the real implementation), where each position is a one in the bloom filter representation. For example, the descriptor 0011100010 in the table of Figure 4.1 creates a path from the root to a leaf that contains the nodes (3,4,5,9). At the end of each path we add a final node, indicated with $ in the picture.

Filters		**(Tree,Interface)**
Bit String	**1s Pos**	
1000100000	(1,5)	(T_1, i_2)
1010000100	(1,3,8)	(T_2, i_4), (T_1, i_2)
0110100000	(2,3,5)	(T_5, i_3)
0011100010	(3,4,5,9)	(T_4, i_6), (T_3, i_2)
0010101000	(3,5,7)	(T_6, i_5), (T_1, i_2)
0000100100	(5,8)	(T_2, i_2)

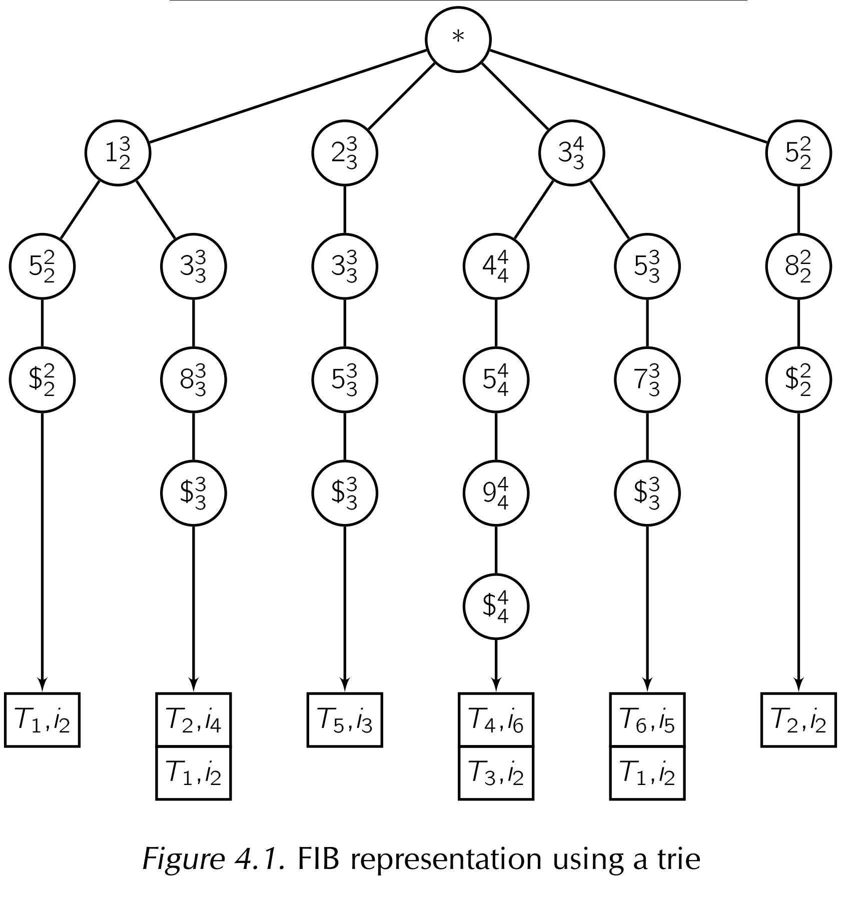

Figure 4.1. FIB representation using a trie

In each node we store also the maximum and the minimum depth reachable from that node. In Figure 4.1, we write the maximum and minimum depths with superscripts and subscripts, respectively.. For example, a node such as 2_3^4 has a maximum depth equal to 4 and a minimum depth equal to 3. This means that the longest path from the root to a leaf that goes through the node 2_3^4 contains 4 nodes, while the shortest one contains 3 nodes. The final nodes are not considered to compute the length of a path.

In addition each node contains a pointer to each children. In the final nodes, the pointer is used to access the list of tree-interface pairs associated to the descriptor. In TagNet, a message is committed to a tree, that does not change in the forwarding process. For this reason, a packet p with a descriptor P that matches a descriptor D in the FIB is forwarded to an interface j only in the case that D has an associated tree-interface pair (T_i, j) and T_i is the tree specified in the header of p.

4.3.1 Find All Subsets

Figure 4.2 presents the algorithm that finds all the descriptors D in the FIB that are subsets of a given descriptor P in the header of an incoming packet. We call this procedure the Find All Subsets algorithm, or FAS algorithm. This is the algorithm used to forward notification packets in multicast. The FAS algorithm takes a packet p as input and finds all the output interfaces where the router can forward p. The function requires also a node n, that is the node to analyze during the execution of the function.

```
 1 void find_all_subsets (packet p, node n){
 2   for(node c : n.children){
 3     if(c.pos == $){
 4       //match found
 5       set_output_interfaces(p, c.interfaces);
 6     }else if(p.descriptor[c.pos] == 1){
 7       find_all_subsets (p, c);
 8     }
 9   }
10 }
```

Figure 4.2. Matching algorithm: find all subsets (FAS)

The algorithm that we propose is quite simple. At line 2 we iterate over the children c of node n, in order to check all the positions stored in these nodes.

If c is a final node (line 3, the node has position \$), we found a match. In this case we need to go through the list of tree-interface pairs associated to c and select the output interfaces for the packet p, if any. This is done in the function *set_output_interfaces*.

In the case that c is not a final node, we need to check if the bit at position *c.pos* in the packet descriptor is set (line 6). If this is the case, we call recursively the function *find_all_subsets* on the node c.

As discussed in Section 3.3.2, a packet can be forwarded using a limited fanout k. The basic algorithm that we just described implements the case where $k = \infty$, and forwards the packet to all the matching interfaces. We have a different version of this matching algorithm that limits the fanout of the packet. This requires a simple adjustment to the code in Figure 4.2. Is it possible to pass the k parameter to the algorithm, and stop the recursion as soon as the algorithm finds k different output interfaces for packet p.

4.3.2 Find Largest Subset

The algorithm in Figure 4.3 finds the descriptor D in the FIB that is the largest subset of the descriptor P in the header of an incoming packet. We call this procedure Find Largest Subset algorithm, or FLS algorithm. This algorithm is useful to forward a request toward a producer node that has the most relevant content with respect to the descriptor in the request. This algorithm can also be used to simulate the longest prefix match semantic required by CCN. If we transform all the hierarchical names in tag set in the way that we described in Section 2.2.4, the FLS algorithm returns the longest prefix that matches the descriptor (or name in this case) in a request packet.

The matching algorithm in Figure 4.3 is conceptually similar to the FAS algorithm. The main difference is that we keep track of the maximum Hamming weight of the descriptor that we match during the execution using the variable *best*. The Hamming weight of a descriptor D represents the number of ones that appear in D, and, in our trie, corresponds to the depth of the final node of D. At the beginning, *best* is equal to 0 (line 22).

As for the FAS algorithm, we iterate over all the children c of a node n. In this case we check the position in c only if the maximum depth reachable thorough c is greater than the best match that we have seen so far (line 4). In fact, if *c.max_depth* is not higher than *best*, there is no possibility to find a better match under c, and so we can skip the node.

In case we get a match (line 7) we need to update the value of *best* and also remember the matching node, which is stored in the variable *best_mach*. This is

```
 1 node find_largest_subset (packet p, node n, int best){
 2   node best_match;
 3   for(node c : n.children){
 4     if(c.max_depth > best) {
 5       if(c.pos == $){
 6         //match found
 7         best = c.max_depth;
 8         best_match = c;
 9       }else if(p.descriptor[c.pos] == 1){
10         node candidate = match(p, c, best);
11         if (candidate != NULL){
12           best_match = candidate;
13           best = best_match.max_depth;
14         }
15       }
16     }
17   }
18   return best_match;
19 }

21 void find_largest_match(packet p){
22   node best_match = find_largest_subset (p, root, 0);
23   if(best_match != NULL)
24     set_output_interfaces(p, best_match.interfaces);
25 }
```

Figure 4.3. Matching algorithm: find largest subset (FLS)

because it may happen that we find more than one match during the execution of the function, and so we need to select the output interface for p at the end of the execution.

At line 9 we check if position *c.pos* is set to one in the descriptor in p. In this case, we visit all the children of *c* with a recursion. The recursive call returns a node only in the case that the function finds a new matching descriptor that has an higher Hamming weight. For this reason, if the node *candidate* is not null, we need to update the variables *best_mach* and *best*.

At the end of the execution, at line 23, if *find_largest_subset* returns a valid node, we can forward the packet to the interface associated to the matching node.

4.3.3 Matching Algorithms Improvements

In this section we introduce some strategies that help us to reduce the memory access and the number of recursions. In this way we can reduce the matching time and increase the throughput of our matcher.

The first strategy that we introduce is related to the Hamming weight of the descriptor D contained in an incoming packet p. The Hamming weight can be efficiently computed using a popcount function. A popcount function returns the digit sum of the binary representation of a given number, which corresponds to the number of bits set to one. In our implementation we use the built-in function of gcc.[2] Before the recursion, in both algorithms, we can check if the Hamming weight of D is higher or equal than the minimum depth stored at node c. If this condition does not hold, there is no possibility to find any match under the node c. This is because all the paths that we can take from c have more ones that D. Since we look for subsets, the number of ones in D should be grater or equal than the ones in the matching path.

The second strategy exploits the same idea. Before the recursion, we compute the number of ones that we did not match yet at position *c.pos* (the position stored in the child c of the node n that we are processing) in D. In other words, we count how many bits are set in D from position $c.pos$ to the end of the descriptor. We add to this value the depth that we reached in the trie, meaning the depth of node c. We call the obtained result the Maximum Potential Match (MPM). MPM indicates how many ones we already matched (depth of c), plus how many we can still potentially match (remaining ones in D) and indicates the maximum Hamming weight of a descriptor that we can match at this point of the execution. The value MPM must be higher or equal to the minimum depth value stored at the node c in order to have a chance to find a matching path under the node c. In the case that MPM is higher or equal to the minimum depth in c we call recursively the matching function, passing the node c as an input parameter.

The two implementation improvements that we described are useful for both the FAS and the FLS algorithm. The implementation improvement that we introduce here can be implemented only in the FLS algorithm. Before the recursion, we can check if the value of MPM is greater than *best*. Only in this case we have

[2]Other Built-in Functions Provided by GCC. https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

the chance to find a better match than the one the we already have. If the condition does not hold we can find only matching descriptors with an Hamming weight smaller than *best*, so we can avoid to visit the node.

4.4 Data Structure and Implementation

What we presented so far in this chapter are the basic ideas behind our content-based matching engine. In fact both the data structure and the algorithms can be improved in different ways. The data structure in Figure 4.1 performs really poorly in a real implementation because this trie requires a lot of space in the memory. This is mostly due to the fact that each node requires multiple pointers, one for each child. Moreover, since we use pointers each node may be located in different parts of memory and, for this reason, the algorithm has a random access pattern to main memory. An unpredictable access to main memory invalidates most of the algorithm for the CPU caching. Since the miss rate in the L1 and L2 caches of the CPU can be high, we need to access main memory frequently. The main memory is slow, and so the performance of the matching algorithm is poor. In this section we present and describe a series of implementation details that help us to improve the forwarding rate of our algorithm.

4.4.1 Memory Footprint Reduction

The first way to improve our algorithms is to reduce the memory usage of our data structure. In particular, a more conscious usage of main memory gives us an higher chance to find the data in the CPU cache, and this can give an important boost to the performance of the matcher. Here we describe three implementation techniques that we use in our implementation to reduce the memory requirements:

1. permute the bits of the descriptors in the FIB according to their popularity.
2. implement the trie using a vector.
3. remove the chains (list of nodes with a single child) in the most depth levels of the trie.

In the following of this section we describe how we apply these implementation techniques on the original data structure, and we analyze the implications and the advantages of each technique.

Bit Permutation

The first transformation that we apply to our trie is to sort the bits in the descriptors contained in the FIB by popularity. Pushing the most popular bits in the first part of the descriptors we create many descriptors with a common prefix. Since we use a prefix-trie, we need to create only one path for all the descriptors that share the same prefix. Less paths to store means less nodes in the trie, and so less memory usage.

Bit Pos	Bit Freq	New Pos
1	2	3
2	1	5
3	4	2
4	1	6
5	5	1
6	0	9
7	1	7
8	2	4
9	1	8
10	0	10

Original Filters	Permuted Filters
(1,5)	(1,3)
(1,3,8)	(2,3,4)
(2,3,5)	(1,2,5)
(3,4,5,9)	(1,2,6,8)
(3,5,7)	(1,2,7)
(5,8)	(1,4)

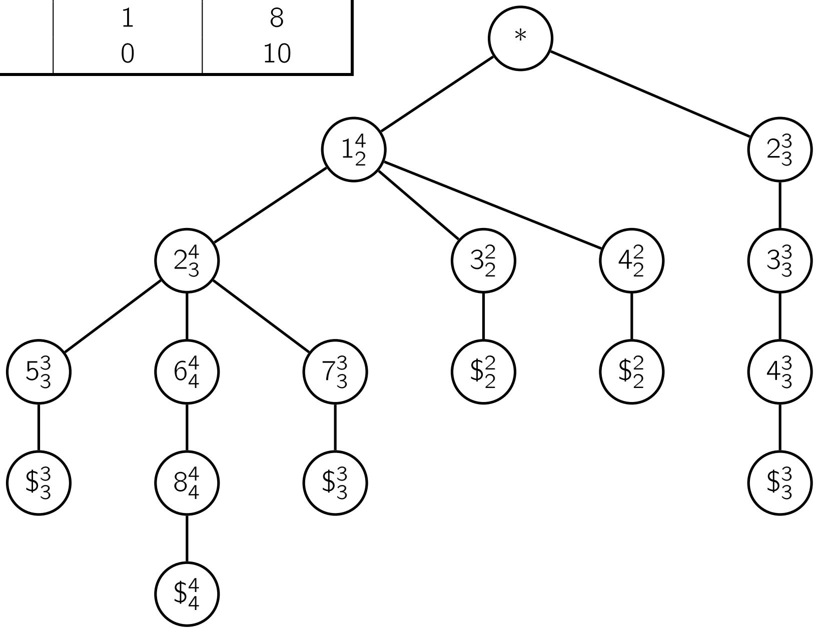

Figure 4.4. Trie compression with bit popularity (the original trie is the one presented in Figure 4.1)

Figure 4.4 shows this transformation applied to the trie in Figure 4.1. The table in the top left part of the picture shows the frequency of appearance of each bit. The new position is computed ranking the bits according to their frequency.

What we obtain is a permutation that we apply to each descriptor in the table. For example, the descriptor (2,3,5) becomes (1,2,5), because the permuted position of 2 is 5, the new position of 3 is 2 and the one of 5 is 1.

The new trie that we obtain is represented in the lower part of Figure 4.4. This trie has 18 nodes, while the original one in Figure 4.1 has 22 nodes. Although the compression obtained in terms of nodes in this small example is not high, in reality the bit permutation is quite effective. In particular, this technique is more efficient when the frequency of appearance of the bits is skewed, so there are only few bits that are set to one with high probability. This is something that can happen easily in reality. For example, if each application adds an application-tag to the descriptors, then the bits related to popular applications will have high probability to be set. The same happens also in hierarchical names, especially with names derived from urls. In a url like name there are just few domain names that can be used as a first name component (e.g com, it, ch, . . .). These components are much more popular than other words, and this popularity is reflected to the bits set for these components in the descriptors.

Another advantage of this technique is that it helps us to skip significant parts of the trie during the matching execution. If an incoming descriptor does not have a popular bit set to 1, the algorithm can skip the entire subtree under such a bit.

The main disadvantage of this technique is that we need to permute also the bits in the descriptor of all the input packets of the matcher. However, this can be done in a fast way, and the algorithm requires a time proportional to the ones that appear in the descriptor of the packet.

Vector Representation

One of the main disadvantages in the trie representation is the usage of pointers. Pointers add a lot of overhead on each node. Each node carries only 3 bytes of useful informations, namely 1 byte for the position, 1 byte for the maximum depth and 1 byte for the minimum depth. All the rest, the pointers in particular, is overhead added by the data structure. One way to reduce this overhead is to consider the trie as a first child-next sibling binary trie, and then transform it in a vector, as described in Figure 4.5. The figure represents the same trie of Figure 4.4. In this representation each node has a single pointer to its first child, which is represented with a plain arrow in the picture, and all sibling nodes are linked together in a list, indicated with dashed arrows.

This data structure can be represented with a vector, where the pointer to the first child of a node is an index or an offset to a particular cell of the vector,

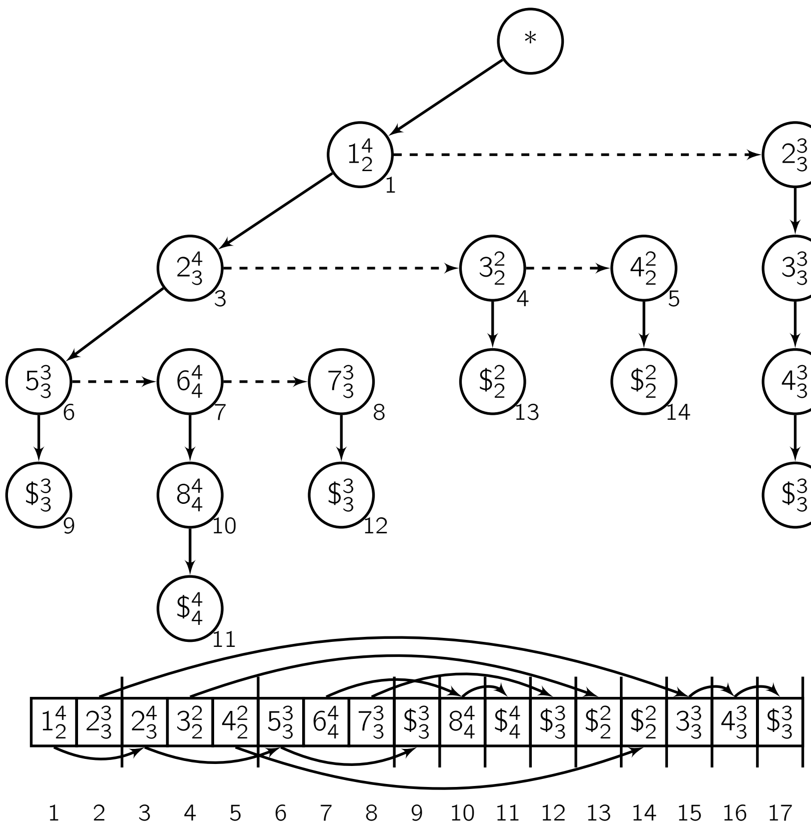

Figure 4.5. Trie represented as a vector

and adjacent nodes are sibling nodes on the trie. In order to recognize the last child of a node (the right most sibling) we add a new field in the node called *last_child*. The lower part of Figure 4.5 shows the vector representation of the trie. Each node has a position, a maximum and a minim depth, as in the original trie. The pointer, implemented with a 32-bit integer, is represented with an arrow and the last_child field is indicated with a vertical bar that separates sets of sibling nodes.

The numbers close to each node of the trie indicates the position of the node in the vector. There are many ways to sort the nodes in the vector, the one in the figure is just an example. Different node layouts define different memory access patterns, which has an impact on the performance of the matching algorithm. Later we analyze two possible node layouts to see which one better fits our matching algorithm.

Chains Removal

The last transformation that we apply to the prefix-trie to reduce the footprint of the data structure is to remove the chains. With the term *chain* we indicate a sequence of nodes with a single child. In this implementation we focus only on the chains that terminate in a final node, namely a node with position \$. A chain is highlighted on the left side of Figure 4.6. When we reach a chain during the matching algorithm, we have the chance to match only a single descriptor, and this is because a chain defines a single path, that corresponds to a single descriptor. For this reason a chain is not useful to navigate among descriptors in the FIB, it simple says yes or no to a particular match. We decided to remove these chains from the trie, and store them in another data structure. The best candidate that we have is the list of tree-interface pairs. In fact, in case of match, we need to access this data structure anyway, and, since we do not expect really long chains, the probability to get a match when we enter a chain is high. The chains with the next hop information are stored in a vector of bytes, as described on the right part of Figure 4.6.

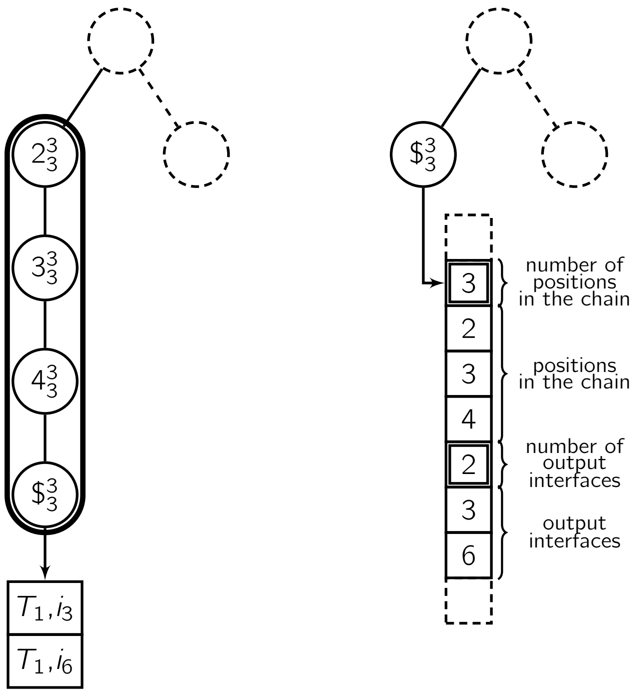

Figure 4.6. A chain in the trie (on the right) and its representation (on the left)

As described by the example, in order to remove a chain from the trie, we remove all the nodes in the chain and we add a final node instead of the first node of the chain, which is 2_3^3 in this case. The new final node has the same maximum and minimum depth of the final node in the chain, which also correspond to the values in the first node. The final node points to a cell in a vector that stores the number of nodes in the chain, so the number of positions that we still have to check in order to match the entire descriptor. In the figure, this value is 3, because we have 3 nodes in the original chain. After the first value we store all the positions that are in the chain. After the positions, we store the information to forward a matching packet. The first value is the number of output interfaces that we can use for the packet, and the following values are the list of the output interfaces.

4.4.2 Implementation Speedup

So far we discussed ways to reduce the size of the trie. In this part of the section we want to discuss some ways to improve the algorithm that we propose.

The first problem that we want to discuss is related to recursion. In fact, both the algorithms that we presented in Figures 4.2 and Figure 4.3 use recursion. Although this is a powerful and useful tool that we can use in programming, when performance is important recursion becomes expensive. For this reason we break the recursion using a stack where we store the pointers to the nodes that we need to visit. Instead to call the recursive function, we push a pointer to the node in a stack and we visit the node later. The execution ends when there are no nodes left on the stack.

Vector Memory Layout

This new implementation highlights another problem described in Figure 4.7. Using the node layout that we select for the vector in Figure 4.5, the algorithm accesses the memory in an almost random way. As described by the example in Figure 4.5, in this layout, we add the nodes of a certain level according to the order of their parents. We call this node layout Sibling Order Layout (SOL), because we visit sibling nodes in order, from the first one to the last one. For example, in Figure 4.5, the children of node 1 are stored before the child of node 2. In particular, the children of node 1 are stored from position 3 to position 5, while the only child of node 2 is at position 15.

In Figure 4.7 we show the evolution of the stack and the memory access pattern when we match the descriptor (1,2,4,5,7) on our trie, using different node layouts. In this picture, the pointers to the next node are represented with a gray arrow, while the sequence of memory accesses are depicted in red. Under each trie we represent the evolution of the stack while we visit nodes on the trie. The values in the stack are the indexes of the cells, indicated over each trie.

The first figure represents the SOL layout. As shown by the picture, this layout creates an almost random access to main memory, due to the usage of the stack for recursion. In the example, when we visit node 1 we have a match (the bit 1 is set in the packet descriptor), so we push node 3 on the stack. Then we need to visit also node 2, because node 1 is not a "last sibling" node, and, since we have another match, we push the node 15 on the stack. At this point the algorithm pops a node from the stack, and jumps to node 15. The position in node 15 does not match the packet descriptor, so the matcher jumps back to node 3. This access pattern continues for the entire execution, and it is costly because it invalidates most of the data cached by the CPU. Notice that also a pure recursive function, like the one in the two algorithms in Figure 4.2 and 4.3, has a similar behavior. This is also another reason way the recursive implementation performs poorly.

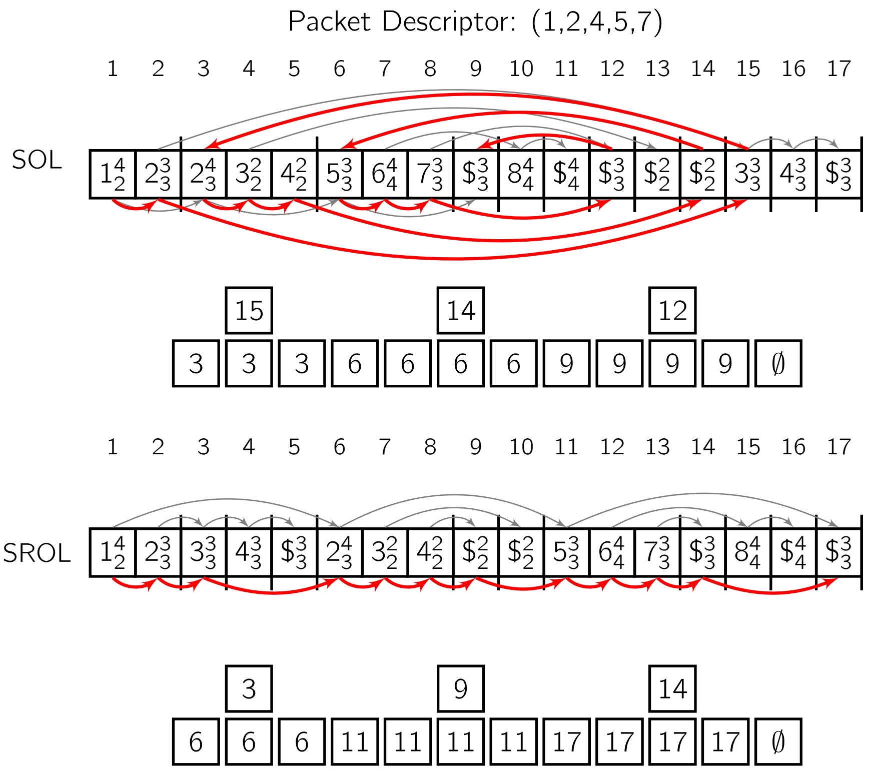

Figure 4.7. Different node layouts and related memory access pattern

In the second layout, we try to solve the problem of the random memory access, or at least minimize it. The layout, presented in the second part of Figure 4.7, is called Sibling Reverse Order Layout (SROL). As in the first layout, also here sibling nodes in the trie are stored in consecutive nodes in the vector. When we need to insert children of sibling nodes, we start visiting the children of the last sibling node, and then we visit the sibling nodes in revers order. As an example, in Figure 4.7 the only child of node 2 is at position 3, while the children of node 1 are stored from position 6 to position 8. In this way, the node where we need to jump is always close to the last node that we visited.

In the example of Figure 4.7 at node 1 we push node 6 on the stack, while at node 2 we push node 3. When we pop the first value from the stack we get node 3, which is the node after node 2 that we are visiting, and, most likely, is already in the CPU cache.

4.5 Locators Based Matching Algorithm

In this section we provide an overview of the TZ routing scheme that we use to route reply and request packets [76]. Here we describe only the most practical version of the algorithm, which is the one that we use in our implementation.

In the TZ routing scheme we forward packets on a tree, so, first of all, we need to cover our network with a routing tree. Each node v in the tree has a weight s_v, which is the number of its descendant nodes in the tree, including v itself. A node v' is considered to be a heavy child of a node v if $s_{v'} \geq s_v/2$. By definition, each node has at most 1 heavy child. If a node is not heavy, then the node is considered to be light. Using these weights we can start to enumerate the nodes in the network. Each node is identified with a number assigned in a depth first order, starting from the root of the tree. The children of a node are visited starting from the lighter one to the heaviest.

The routing information to store on each node v is represented by the tuple $(v, f_v, h_v, \ell_v, P_v)$, where:

- v, is the number assigned to the node, and identifies the node on the tree.
- f_v indicates the identifier of the largest descanted of v.
- h_v is the identifier of the heavy child of v, if it exists, otherwise is set to $f_v + 1$.
- ℓ_v is the light level of v. This value indicates the number of light nodes that we traverse on the path from the root of the tree to v, including v in the counting if v is a light node.

- P_v is an array with 2 components. $P_v[0]$ is the interface to the parent node of v, while $P_v[1]$ stores the interface to the heavy child of v.

Now that we have all the information to route a packet on the tree, we need to create the labels, the so called TZ-labels. A TZ-label is what we use to forward the packets to the next hop. We define the path $\langle v_0, v_1, \ldots, v_k\rangle$ to be the path from the root r of the tree to a node v, where $r = v_0$ and $v = v_k$. i_j, with $1 \leq j \leq \ell_v$, indicated the j-th light node on this path. We define the array $L_v = [\mathit{ifx}(v_{i_1-1}, v_{i_1}), \mathit{ifx}(v_{i_2-1}, v_{i_2}), \ldots, \mathit{ifx}(v_{i_{\ell_v}-1}, v_{\ell_v})]$ as the array of the interface numbers that, from a node i_{j-1}, lead to a light weight node i_j on the path from the root to v. The label associated to each node is then defined as $\mathit{label}(v) = (v, L_v)$. This is the label to put in the header of a packet to indicate the destination of such packet.

The forwarding process works as follow. A packet with the label (v, L_v) arrives at node w. In the case $v = w$ the packet reached is destination. Otherwise we need to check if v is in the set of the descendant node of w, in other words we check if $v \in (w, f_w]$. If v is not a descendant of w, we need to forward the packet to the parent of w, using the interface $P_w[0]$. In the case that we do not send the packet to the parent of w, we check if v is a descendant of the heavy child of w, by checking if $v \in [h_w, f_w]$. If this is the case, we send the packet to the interface $P_w[1]$, that indicates the interface to the heavy child, otherwise we need to decide which is the right light child of w to select as next hop. This can be easily done using the array L_v in the label. In particular $L_v[\ell_w]$ indicates the interface to the right light child, where ℓ_w is the light level of w.

Using this algorithm, the size of the label that we need to carry in a packet is $O(\log_2 N)$ words, where N indicates the number of nodes in the tree. This is because the size of the array L_v is equal to ℓ_v for each node v, and it is easy to see that ℓ_v is at most $\log_2 N$. Every time that we go from a certain node to one of its light children, the number of nodes in the remaining subtree is reduced by at least a factor 2. This is because the definition of light child: a light child has always less than 1/2 of descanted of its parent. For this reason ℓ_v can not be higher than $\log_2 N$. Thorup and Zwick propose also a more compact version for the labels, and we use this more tight version in our implementation. With this new encoding, the forwarding algorithm does not change significantly. Using the compressed scheme, our labels, computed on the AS-level topology, are at most 46-bits long, and so we use 64 bits to represent them in the header of our packets.

4.6 Evaluation

In this section we evaluate our forwarding engine. In our evaluation we use 3 different workloads to create the router FIB, in order to show that our implementation works in different settings. The first workload that we use is the one generated during the experiment reported in Figure 3.19 of the previous chapter. This workload contains more than 63 million unique descriptors and we identify it with the label 63M. The second workload that we use is a sample of 10 millions descriptors that we extract from 63M. We indicate this workload with the label 10M. We use 10M because 10 million is the average size of the workloads used to test other matcher in other ICN architectures. We also want to show that, even if 63M is more than 6 times bigger than 10M, the difference in the throughput between the two workloads is smaller, and so the matching time does not grow linearly with the size of the FIB. The last workload that we use is labeled 10M-NSDI and is composed by almost 10 million entries. This workload was created by Wang et al. to test their GPU-based matcher published in the NSDI conference [82], and was used also in other works [63, 81]. The workload is composed by prefixes and not by descriptors. We use this workload to show that our algorithm performs well also in the case where we have to emulate the CCN hierarchical names.

We generate the descriptors in the request and notification packets by taking a descriptor from the FIB and adding some random bits to it. In particular in the construction we add up to 14 bits to each descriptor, that are the equivalent of 2 tags, since we use 7 hash functions to represent each tag in our bloom filters.

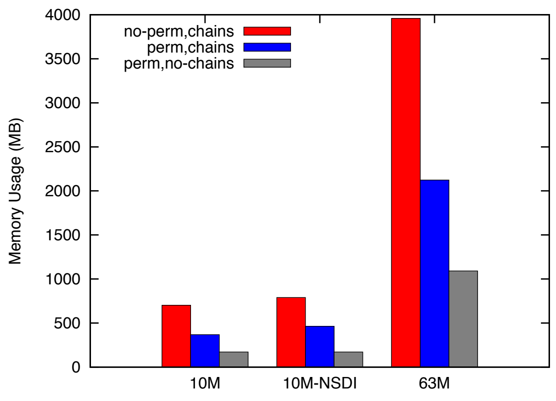

Figure 4.8. Prefixes compression scheme

We start the evaluation showing the memory usage of the trie for all the workloads, using different compression strategies. Figure 4.8 shows the results of our evaluation. All the results refer to the trie in the vector form. In the plot, the red bar indicates the memory required by the trie with no compression (no-perm, chains), the blu bar indicates the memory usage when we apply the bit permutation (perm, chains), and the gray bar represents the memory needed when we also remove the chains (perm, no-chains). The plot shows that the bit permutation is quite effective on all the workloads, both in case of descriptors and in case of prefixes. In particular we reduce the FIB size by 48% in case of 10M, by 42% in case of 10M-NSDI and by 46% with 63M. Also the removal of the chains is effective in all the cases. In the end, the size of the FIB for 10M and 10M-NSDI is 171MB, while for 63M we need 1.06 GB.

Table 4.1 reports the average matching time, computed over 4 million packets, for the two different node layouts that we proposed in this chapter. The times are in microseconds (μs). In particular, the table shows the matching time required by the two content-based matching algorithms for all the workloads. This times are obtained using a single thread.

	Find All Subsets		Find Largest Subset	
	SOL	SROL	SOL	SROL
10M	43.3	39.4	28.9	25.1
10M NSDI	69.8	66.9	41.8	36.4
63M	97.8	93.7	69.4	61.9

Table 4.1. Matching time for different nodes layouts (in microseconds)

What is immediately clear from the table is that the SROL layout is always faster than the SOL layout. In the worst cases, represented by the FAS algorithm using 10M-NSDI and 63M, we gain only 4.1%. However, in the case of the FLS algorithm we always gain more than 10%, with a peak of 13.1% in the case of the 10M workload.

These results are particularly important, as they effectively measure the latency of our matcher. In fact, the values reported in the table are the average latency experienced by each packet. In particular, the latency of our implementation is the one of the SROL layout, so the values reported in the second and fourth column. The latency should be lower than 100μs [82], and our implementation is able to meet this requirement for all the workloads.

At this point of the evaluation we want to test the scalability of our implementation with respect to the number of threads. We run our code on a machine equipped with two Intel Xeon E5-2670 v3 processors, each one with 12 cores, and 2.30GHz clock frequency. The machine has 64GB of RAM.

In Figure 4.9 we show the throughput of the FAS algorithm varying the number of threads. The throughput is measured in Kilo packets per second (Kpps). The plot shows that all the three workloads scale almost perfectly increasing the number of threads. Every time we double the number of threads we gain on average 80.8% in throughput. The minimum gain that we achieve is 57.1%, when we go from 16 threads to 32 threads with 63M. This is due to the fact the our machine has only 24 cores, so, in order to run 32 threads, we need to use hyper-threading. Using the FAS algorithm our router can forward 499,3Kpps with 10M, 307,7Kpps with 10M-NSDI and 183,5Kpps using 63M.

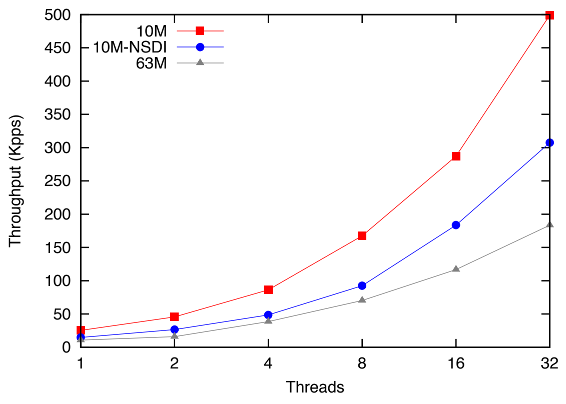

Figure 4.9. FAS algorithm: scalability with multiple threads

Although 10M and 10M-NSDI require the same amount of memory, there is a difference between the performance achieved using the two workloads. This is due to the fact that 10M-NSDI has more paths in the top part of the trie with respect to 10M, due to the fact that we simulate CCN hierarchical names. In this case, the algorithm needs to explore more nodes and it requires more memory accesses. Another consideration that we can derive from Figure 4.9 is that, although 63M contains 6 times more filters than 10M, the run with 10M is only 2.45 times faster on average. This confirms the idea that our algorithm works in less than linear time and scales with the size of the FIB.

In Figure 4.10 we show the same analysis for the FLS algorithm. Also in this case we can see that implementation scales well with the number of threads. On average we gain 83.1% in throughput every time we double the number of threads. Like for the FAS algorithm, the minimum gain that we obtain is when we pass from 16 to 32 threads using 63M. In particular, in this case, we gain 59.3%. The throughput in the case of FLS is almost twice the throughput of the FAS algorithm. Our matcher processes 914,9Kpps with 10M, 602,8Kpps in case of 10M-NSDI and 272,7Kpps using 63M.

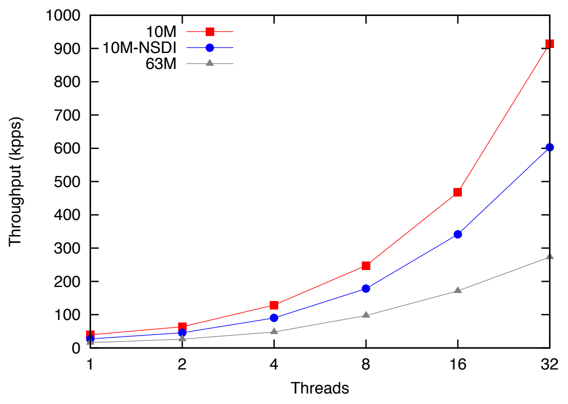

Figure 4.10. FLS algorithm: scalability with multiple threads

So far we observed only the performance of a single algorithm. In particular we saw how many packets of a certain kind our matcher can process every second. In reality our matcher is able to process different kind of packets at the same time. As we already said, we use the FAS algorithm to forward the notification packets, the FLS to forward the request packets and the TZ-labels to forward the request and reply packets. In the two following experiments we create a traffic-mix with all these types of packets. The matcher decides every time which algorithm to use, according to the type of the incoming packet.

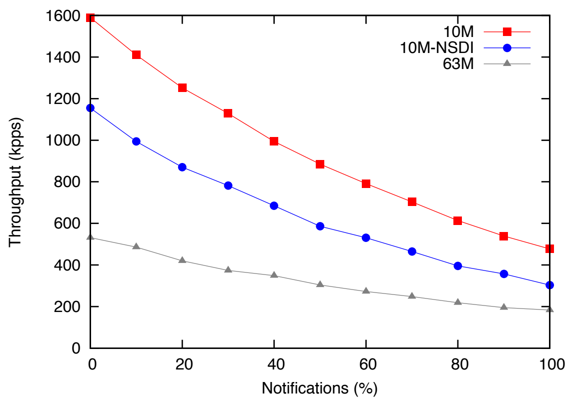

Figure 4.11. Throughput varying the percentage of notification packets

In Figure 4.11 we show the throughput varying the percentage of push and pull traffic. The percentage of push traffic is indicated on the x-axis. The rest is pull traffic. Push traffic is composed by notification packets, one for each flow. Pull flows are composed by two packets: a request packet to forward using the FLS algorithm, and the corresponding reply packet that is forwarded using the

TZ-label.

The plot shows that notifications (push traffic) have an impact on the performance of the matcher, especially in the case of 10M and 10M-NSDI. For example, in case of 10M the matcher forwards 1.58 Million packets per second (Mpps) when there is no push traffic, and the throughput decreases to less than 500Kpps in case we have only notification packets. This result is consistent to want we measure in the Figure 4.9.

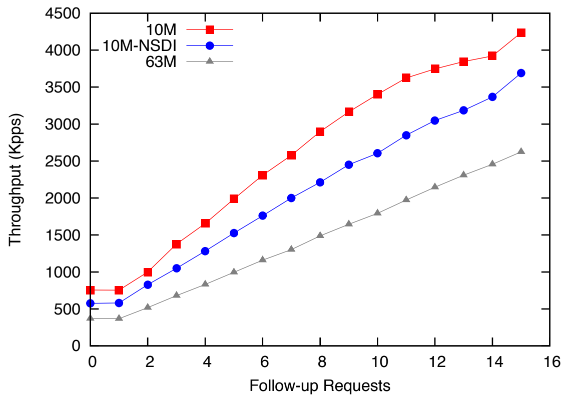

Figure 4.12. Throughput varying the number of follow-up requests forwarded using the locators.

In the last experiment we set a more realistic traffic-mix, where we implement the communication flow described in Section 3.3.3. In this case we fix the percentage of push traffic and we create a workload where 30% of the traffic is push, while the rest is pull. As in the previous experiment, push traffic is composed by notification packets, one per flow. Instead, each pull flow contains a variable number of packets. In each flow we have a request packet to forward with the FLS algorithm and a series of follow-up requests that we forward using TZ-labels. The number of follow-up requests is randomly selected in the range between 0 and a maximum value, which is indicated on the x-axis of the chart in Figure 4.12. We vary this parameter during the experiment. As in the previous experiment, a pull flow contains a reply packet for each request, no matter which algorithm we use to forward the request.

What is immediately clear from Figure 4.12 is that, even if we forward only few requests using locators in each pull flow, the throughput of our matcher receives an important boost, and we can easily forward few millions of packets per second. In particular our matcher can achieve a throughput of 20Gbps with an average packet size of 952.4 bytes in the case of 63M, 677.4 bytes for 10M-NDSI and 589.8 bytes for 10M.

Chapter 5

Transport and Congestion Control

In this chapter we introduce a transport protocol with congestion control for ICN. The congestion control is implemented at the receiver node and it is able to control flows on multiple paths without using a priori knowledge of the network. Combining our congestion control protocol with a distributed forwarding strategy (also described in this chapter) implemented on each router, we are able to maximize the usage of the available bandwidth in the network, minimizing the transfer time of a content object.

The protocols that we describe in this chapter are designed for CCN, but they can be easily used in our architecture. The congestion control does not require any in-network state, meaning does not relay on the PITs, and all the required state to control the transmission is handled by the receiver. This means that the congestion control can work properly also in TagNet.

For the forwarding strategy, the porting is a bit more involved. The algorithm, in fact, is based on the assumption that each data packet comes back to a router from the interface where the router sent the related interest. This is always true in CCN, where the flow balance between interest and data packets is always guaranteed, since a data packet follows the revers path of the corresponding interest. This is also true in TagNet, when we use only one level in the routing protocol: the reply packet always follow the reverse path of the request, which is the only one on a tree. Unfortunately, when we use more than one layer in the routing (like in the case of inter- and intra-domain packets forwarding) a reply packet may go through a path that is different from the one used by the related request. However it is possible to modify the forwarding process in the hierarchical setting in order to achieve flow balance also in TagNet. This requires to store more labels in the source and destination stack of the packet. In particular we need to keep track of the trees used at the local level to send the packet to the appropriate gateway router. With this information, we can always receive a reply packet on the reverse path of the corresponding request, and so we can use the forwarding strategy proposed in this chapter.

The Chapter is structured as follows: Section 5.1 presents related work both in traditional networks and in ICN. Section 5.2 describes the transport protocol that we propose, and, in Section 5.3, we extend it in order to control multiple paths for the same communication flow. Section 5.4 presents the distributed forwarding strategy implemented on each router in the network. In Section 5.5 we evaluate the proposed congestion protocol and the forwarding strategy through simulations.

5.1 Related Work

Multipath congestion control is widely studed in literature, both from the theoretical and the engineering point of view. All the proposed algorithm are based on TCP.

One transport protocol that exploits multiple paths is mTCP [88]. mTCP establishes an independent connection for each path used by a flow, called subflow. Each subflow has its own congestion window which is regulated in a similar fashion to the standard TCP. mTCP uses multiple controllers because, as the authors show in the paper, a flow that uses multiple paths and is controlled with a single instance of TCP may get lower bandwidth than one that uses a single path.

mTCP has problems in controlling bottlenecks that are shared among flows. This is due to the fact that each path has its own controller. EWTCP [44] introduces some correlation among the subflows, in order to overcome the problems of mTCP. Each subflow is associated to a weight that reflects the network resources used by the subflow at any time. The congestion window size adjustment is proportional to the weight assigned to each subflow.

MPTCP [84] is the standardize version of multipath TCP, that improves previous proposed algorithms using the network resources in a more efficient way.

There is also a lot of theoretical work that studies the stability of multipath congestion control [43, 50].

All these approaches listed above are not directly usable in ICN, because in ICN there is no a real connection between the receiver and all the sources used to retrieve the content. The receiver node does not know a priori the nodes that will reply to its requests. There may be multiple publishers for the same content, on-path caches may send requested data to the receiver and the set of nodes that reply to the user may change over the time. Basically, is not possible to set up any state for each path that will be used in the transfer of a content object, because these paths are unknown. For these reasons in ICN the congestion control needs to be located at the receiver, and not at the senders as in standard TCP. The receiver, in fact, is the only node that, at any time, has all the information about the ongoing transmission.

Clark et al. present one of the first proposals for a receiver-driven congestion protocol [25]. This is a rate-based transport protocol, where the retransmission timer is handled by the receiver node.

WTCP [72] has the goal to reduce the problem of TCP in wireless environments. In wireless transmission, many packet losses are due to the channel itself, which is not as reliable as a wire connection, and not because of congestion. This leads to poor performance of TCP that interprets each packet loss as a sign of network congestion. WTCP delegates some of the monitoring function to the receiver, that can adjust the sender window size. The receiver adjusts the transmission rate of the sender, sending the correct rate in the ACK messages. Similar ideas are used by other proposals, such as TFRC [31] and TCP-Real [78].

A receiver-driven congestion control that keeps all the state at the receiver is RCP [46]. This transport protocol works in a similar way to TCP, but the receiver node is responsible of all the protocol functionalities, like congestion control and loss recovery. Kuzmanovic et al. analyze in detail RPC, highlighting the improvements and the vulnerabilities of a receiver-driven control with respect to the standard TCP [52].

Looking for specific transport protocols developed for ICN there are few proposals available. One of the first proposed transport protocols for CCN is ICP [11], which is the starting point of the algorithm that we introduce in this chapter. We describe ICP during the presentation of our congestion control.

Saino et al. propose CCTCP that sends anticipated interests to estimate the RTT of further requests [70]. Each interest packet carries a set of chunk identifiers that the receiver will require in the immediate future. If a router (publisher node or cache) has the data required by one of these interests, the router puts a time stamp in the packet. In this way the receiver can have a good estimation of the RTTs for the next set of interests, and the controller can set a correct timer for the interest expiration.

A more recent work is ECP [68]. In this proposal the congestion is detected at middle routers. When a router detects congestion, it notifies the receiver using special NACK packets. The receiver reduces the size of the congestion window when it receives the signal for a congestion event.

Another approach proposed for CCN, is to use hop-by-hop congestion control. The main idea in this case is to control the rate of interest packets, meaning reduce the transmission frequency on each hop, introducing some delay, in order to prevent congestion. Examples are HR-ICP [12] and the protocol proposes by Wang et al. [80]. These distributed algorithms can be used in conjunction with another receiver-driven congestion controller, as in the case of HR-ICP.

5.2 Receiver-Driven Congestion Control

In this chapter we present a receiver-driven congestion control that allows efficient file transfer, controlling the requests rate at the receiver and avoiding network congestion. We design our congestion control with three goals in mind:

- Reliability: The protocol needs to guaranties that the receiver gets the required content. In case some packets get lost during the transfer, the receiver needs to issue a new requests for the missing content.
- Efficiency: Our congestion control is designed to utilize all the resources available in the network, in order to minimize the data transfer time. The congestion control can use multiple paths, in order to download different parts of the same content from multiple sources and use more bandwidth.
- Fairness: The congestion control fairly allocates the available bandwidth among flows that share the same path.

The protocol that we propose is called *Remote Adaptive Active Queue Management* (or RAAQM) [13, 14] and is an evolution of the Interest Control Protocol (ICP) [11]. RAAQM uses an Additive Increase Multiplicative Decrease (AIMD) congestion window to control the rate of the requests that a user is allowed to express. The window is controlled by the receiver node that maintains all the necessary state.

The receiver maintains a window of size W that specifies the maximum number of outstanding requests, meaning the number of requests that a node can issue before receives the corresponding data packets. Every time the user receives a data packet for an outstanding request, the controller increases the window size W by η/W. The window grows by η, which is set to 1 in our implementation, every time that the user receives all the data for an entire window of requests. Figure 5.1 describes the increment of the window.

When a packet gets lost or congestion in the network is detected, the controller decreases the size of the window. The window size is multiplied by a decreasing factor $\beta < 1$. The factor β can be adjusted in order to obtain a

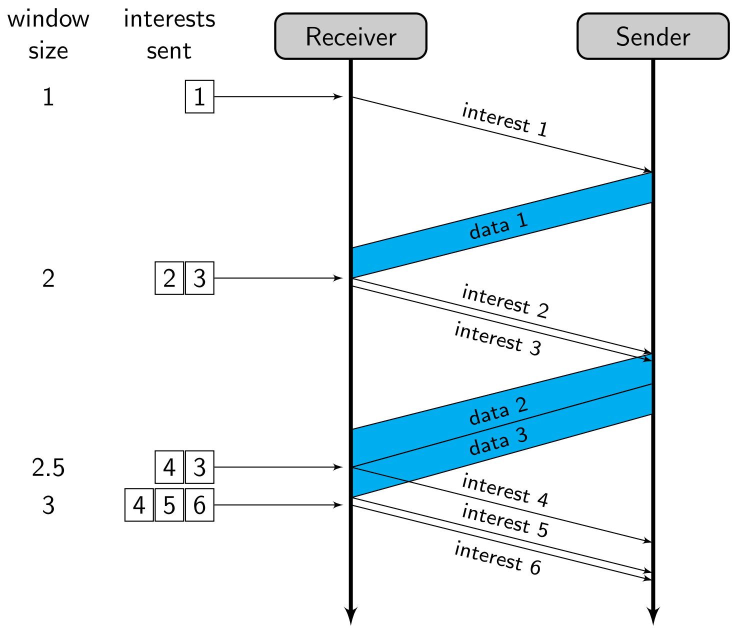

Figure 5.1. Window evolution: additive increase

congestion control that is more o less aggressive. Figure 5.2 shows how the congestion control decrease congestion window size.

5.2.1 Trigger Window Decrease

Define a way to trigger the window decrease can be complicated. If the controller decreases the window too often, the congestion control may be too aggressive. The results is a waste of network bandwidth and the transfer time increases. In the opposite case, the congestion control may not work properly and create congestion.

In ICP the window decrease is deterministically. ICP relies on a timer τ associated to each request. The timer τ is computed as a function of the Round Trip Time (RTT) estimated at the receiver over a set of requests. In case the data packet is not received within the time τ the controller decreases the window W and issues again the expired request. In this way, if the request or its related data packet got lost or dropped some where in the network, the receiver can always get the data. This guarantees the reliability of the congestion control protocol: the receiver will eventually receive the entire content.

Define a simple adaptive timer τ that reflects the network congestion con-

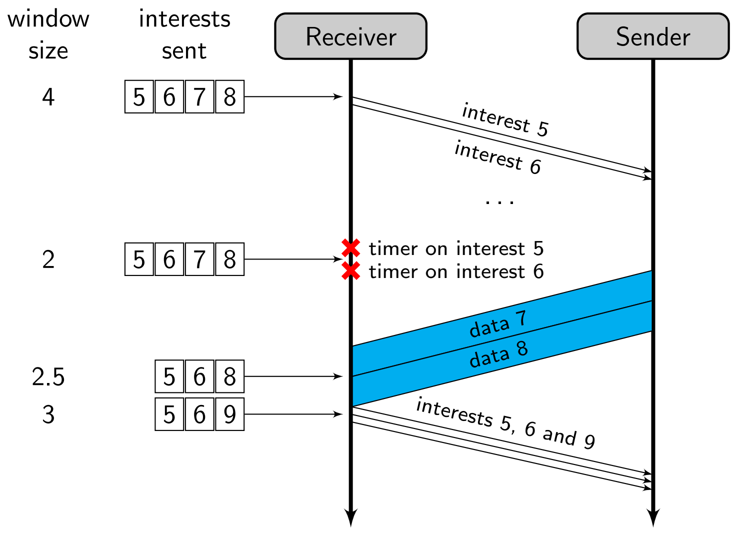

Figure 5.2. Window evolution: multiplicative decrease ($\beta = 0.5$)

dition, such as the one used in ICP, is not easy. The timer, in fact, is subject to estimation errors, due to the RTT variations. Furthermore, in case of multiple paths, the RTTs may vary significantly from one request to the other, and a simple timer is not sufficient to control the window size. In this scenario in fact, short RTTs, related to the requests satisfied by the path with more available bandwidth, force the congestion control to set small τ. This cause a timer expiration for all the requests forwarded on the other paths, that translate in a small window size and poor transfer rate.

RAAQM uses a probabilistic window decrease, aiming to control the queue size on the bottleneck links. The idea is to anticipate the window decrease to prevent a possible congestion in the network, and, at the same time, to reduce window size oscillations. The idea to control the queue delay at the bottleneck comes from Active Queue Management (AQM) algorithms, such as RED [32] or the more recent CoDel [58]. The goal of these algorithms is to reduce the bufferbloat problem, by probabilistically drop packets from the buffer queue of routers. Ardelean et al. propose a way to control queue delay remotely, using RTT estimation at the receiver [5]. RAAQM uses the same ideas, but the probability to reduce the congestion window size is adaptive, and it depends on the RTT estimation.

For every request the receiver node computes the instantaneous RTT, indicated with $R(t)$, that measures the time between the transmission of the request and the reception of the corresponding reply packet. Using this data the controller computes the minimum and the maximum RTT, denoted as R_{min} and R_{max} respectively, over a history of samples. In our implementation the controller keeps track of the last 30 measured RTTs. RTTs of retransmitted packets are not used in the estimation of R_{min} and R_{max}. $R(t)$ and the estimation of R_{min} and R_{max} are used to compute the *decrease probability* $p(t)$, which is the probability to decrease the window size at any given time.

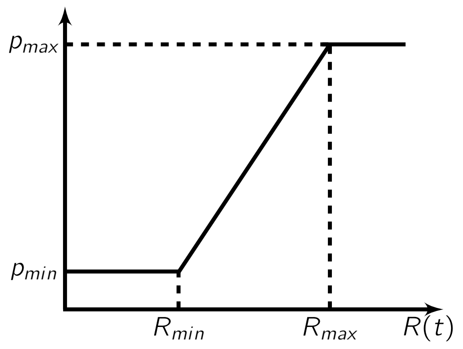

Figure 5.3. Window decrease probability function

Figure 5.3 presents the function $p(t)$: it is a monotonically increasing function of $R(t)$ that goes from p_{min} to p_{max}, which is a value less or equal to 1. $p(t)$ is equal to p_{min} when $R(t) \leq R_{min}$, and it is equal to p_{max} when $R(t) \geq R_{max}$. Our congestion control defines $p(t)$ as follows:

$$p(t) = p_{min} + \Delta p \frac{R(t) - R_{min}(t)}{R_{max}(t) - R_{min}(t)} \tag{5.1}$$

where $\Delta p = p_{max} - p_{min}$, $R_{min}(t)$ and $R_{max}(t)$ are R_{min} and R_{max} estimated at time t. Using $p(t)$ RAAQM can adjust the size of the window at the receiver, increasing or decreasing the load on the bottleneck queue.

The size of the window over the time, indicated with $W(t)$, is described by the following equation:

$$\dot{W}(t) = \frac{\eta}{R(t)} - \beta W(t) p(t - R(t)) \frac{W(t - R(t))}{R(t - R(t))} \tag{5.2}$$

The value $\frac{\eta}{R(t)}$ represents the incremental factor in the AIMD controller. This value is inversely proportional to $R(t)$: with higher RTTs the congestion window is increased less. $\beta W(t)$ is the multiplicative decrease factor, which is applied to the window with rate $p(t - R(t)) \frac{W(t-R(t))}{R(t-R(t))}$. In other words, each reply packet that comes back to the receiver node generates a window decrease with probability

$p(t-R(t))$, which is computed using the RTT estimation of the previous packet, received at time $(t-R(t))$.

5.3 Multipath Extension

The algorithm described so far works correctly on a single path. In an ICN network there are multiple caches that can be used to retrieve the content, or multiple sources for the same content can be exploited. Therefore, the communication in ICN is inherently multipath. We want to extend RAAQM to control multiple paths in the same flow, in order to fit the ICN communication model. The main challenge is to control possible multiple bottleneck using a single controller.

Before we start the description of the multipath scenario we need to define the concept of route. A *route* is a sequence of routers that connect a data source, a cache or a publisher node, to the user. A route is indicated with a label composed by the identifiers of the nodes, ordered from the data source to the users, as represented in Figure 5.4. In the figure node e is the producer node, while node a is the receiver. The figure shows two routes, labeled $\{e,d,c,b,a\}$ and $\{e,d,f,b,a\}$, from the producer node, and a third one, with label $\{f,b,a\}$, from the cache at node f. The identifier of each router can be the MAC address of the node or some kind of special names.

There are at least 3 ways to extend our controller:

1. we can use the controller without any change, with a single congestion window and a single RTT estimator,
2. we can use multiple controllers, with one congestion window and one RTT estimator per each route,
3. we can use a single window, but use a different RTT estimation for each route.

In the first case we use a single congestion window and a single RTT estimation. In this way, the decrease probability $p(t)$ is computed over all the RTT samples received by the receiver node, even if they refer to multiple routes. This amounts to compute $p(t)$ over a global queue delay, which is the average queue delay over multiple routes, with possibly very different delays. In this scenario R_{min} represents the RTT estimated for the fastest path, while R_{max} represents the RTT of the most congested path. This may easily lead to instability, because the congestion control receives many RTTs that are close to R_{min} and just few or even

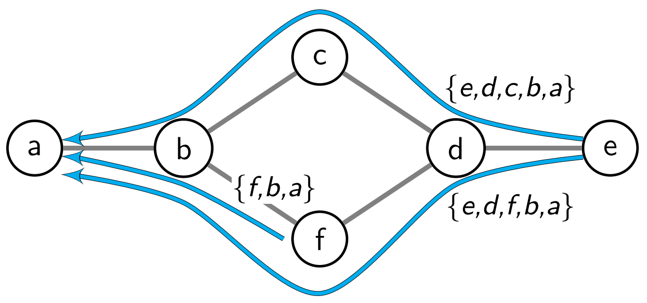

Figure 5.4. Different routes and route labels

none close to R_{max}, because the slower path is congested. This is interpreted by the controller as if the window size is to small, because most of the RTTs are close to R_{min}. As a reaction the congestion control increases the window size, creating more congestion.

The second idea is to maintain one controller for each path. This could be the best approach, but, as we discuses in the related work section, it is infeasible in practice. In ICN the number of routes is unknown a priori and they may change during the file transfer.

The last option, which is the one that we adopt, is to use a single window to control all the routes, but to estimate the RTTs, and so the decrease probability $p(t)$, per route. In order to figure out which route is used by a packet the congestion control uses the labels introduced in Figure 5.4. The receiver collects RTTs separately for each route and, using these samples, estimates the R_{min} and R_{max} of each route. Using all this information the controller can compute a decrease probability $p_r(t)$ that is related to a particular route.

This version of the controller overcomes the problems of the other two solutions. The controller is stable, in fact $p_r(t)$ is not computed over the average RTT, but is related to a particular route and each route can decrease the size of W independently. At the same time, the congestion control does not need to know all the routes in advance, but RAAQM can discover the routes during the file transfer. In our implementation we compute the decrease probability $p_r(t)$ for a route r only if we see at least 30 packets coming from that route.

Using $p_r(t)$, which is computed like in Equation 5.1 for each route, the window evolution for the multipath scenario is described by the following equation:

$$\dot{W}(t) = \frac{\eta}{R(t)} - \beta \sum_{r \in \mathcal{R}} p_r(t - R(t)) W(t) s_r \frac{W(t - R(t))}{R(t - R(t))} \tag{5.3}$$

The incremental factor do not change with respect to Equation 5.2. Instead, the decremental factor is multiplied by the sum of all the decreasing probabilities p_r, computed for each monitored route. The set of monitored routes is indicated with $\mathscr{R}$. $W(t)s_r$ indicates the part of the requests in the congestion window forwarded on route r at time t. The value s_r indicates the *split ratio*, which is the percentage of outstanding requests routed on a certain route, according to the forwarding decisions taken by each node in the network. Notice that the receiver does not need to know s_r, because it simply computes p_r when it receives a packet from a certain route r.

5.4 Forwarding Strategy

The congestion control that we described in the previous sections is able to prevent congestion in the network and handle multiple routes for the same flow. However RAAQM alone does not guarantee good performance in terms of bandwidth utilizations, especially in the case of multiple paths. RAAQM tends to set the transmission rate in order to avoid congestion on the links with smaller capacity. Instead we want to use all the routes at their maximum capacity, to minimize the transfer time.

An aspect that we did not take into account so far is the split ratio. The split ratios are a crucial component for the window size evolution, and, as a consequence, for the transfer rate, as described in Equation 5.3. The split ratio effects how request packets are forwarded in the network at each hop, and this has an impact on the final utilized bandwidth. The split ratio, in essence, is determined by the forwarding strategy used at each node to forward requests. A forwarding strategy can select an output interface for a request packet randomly or with a more sophisticated process. In any case, this is a distributed process that is unknown to the receiver node, so is not part of the congestion control itself.

A good forwarding strategy may push more traffic toward the links with more capacity or the to ones with a lower RTT. In this way the network balances the traffic over different paths, according to their real capacity, and it is possible to use the available bandwidth in a better way. There are some proposals that tries to solve this problem in CCN. CCN itself proposes a layer in its protocol stack, called Strategy Layer, that has exactly the purpose to select the best interface where to forward each interest. Yi et al. [87] propose a mechanism that uses probe packets to infer the quality of a path behind an interface. Using this technique each router ranks the interfaces and uses them according to this rank. This forwarding strategy can be integrated in a routing scheme [86]. Another example of forwarding strategy proposes to forward interests off-path, meaning on a direction that may not lead to the source of the content, in order to explore the nearby caches [24]. Also this mechanism is based on a kind of ranking for the interfaces.

Here we propose our forwarding strategy, that we use in the network in combination to the RAAQM protocol. Since this forwarding strategy was initially proposed for CCN, in this section we use terminology that is CCN specific. We remind that an interest packet in CCN terms is semantically equivalent to a request packet in TagNet, as well as a CCN data packet is semantically equivalent to a reply packet in TagNet. We want also to remind that, as we described in the introduction of this chapter, it is possible to implement this forwarding strategy also in TagNet, with simple modifications.

The forwarding strategy that we propose is completely distributed, requires only local knowledge, does not require any additional traffic and is optimal over the time [14]. This forwarding strategy works under the assumption that, when we send an interest to an interface, we get the data packet on the same interface, since a data packet always follow the reverse path of the related interest. Under this assumption we can count the number of *Pending Interests* (PI) on each interface of the router for a particular prefix. PI is associated to an interface and to a prefix. It is important to associate the PI to a prefix, and not just to an interface, because a router can send interests that match different prefixes to the same interface, but they will have different RTTs, in fact, most likely, they will follow different paths. The values of PI are stored in the FIB of the router.

In CCN the information related to the pending interests is available in the PITs. It is important to notice that the information that we store in the FIB to implement our forwarding strategy is different from the information in the PIT. PITs store information at the chunk level, meaning that, at every instant, each PIT has the number of pending interests for a particular chunk of a particular content. This data are not relevant for our algorithm, because the forwarding decision are made only on prefixes.

The PI gives us a measure of the status of the route behind a certain interface for a particular content. If the number of PI is high, the route may be congested, while if the number of PI is close to 0 there is available bandwidth to use. A router sends an interest to an interface with a probability that is inversely proportional to the related PI.

Figure 5.5 describes the algorithm that we developed for our forwarding strategy. At each reception of an interest (line 1) or data packet (line 9) we update the value of PI associated to a particular interface and prefix. We simply increase PI when we send an interest (line 15) and we decrease PI when we get

```
1  void process_data_packet(string name, int face_in){
2    if (!PIT_miss) then{
3      fwd_data_packet(face_out);
4      PI_dec(face_in, name);
5      ifx_weight_update(face_in, name);
6    }else{
7      drop packet;}}

9  void process_interest_packet(string name, int face_in){
10   if (Cache_miss && PIT_miss) then{
11     ifx_weights_list = FIB_lookup(name);
12     ifx = rand_wighted_selection(ifx_wights_list);
13     forward_interest(ifx);
14     PIT_update(name, face_in);
15     PI_inc(ifx, name);
16     ifx_weight_update(ifx, name);
17   }else{
18     follow CCN standard procedure;}}

20 void fx_weight_update(ifx, name){
21   avg_PI(ifx,name) = $\alpha$*avg_PI (ifx,name)+ (1 − $\alpha$) PI(ifx,name);
22   weight(ifx,name) = 1 / avg_PI(ifx,name);}
```

Figure 5.5. Forwrarding strategy algorithm

a data packet (line 4). Every time that we update the instantaneous value of PI associated to a prefix on a certain interface, we also recompute the weight associated to that prefix, calling the function at line 20. In this function we compute a weighted moving average of PI, indicated with *avg_PI*, that we use in order to have a more stable value of PI. The α parameter of the moving average is set to 0.9. The weights associated to each prefix on an interface are computed as the inverse of *avg_PI*. These weights are used at line 12 to select an output interface for an interest, using a random weighted algorithm. At start time all the interfaces have the same weight, so the forwarding process is uniform on all the output interfaces. Since we use a random decision process to select the output interface, all the interfaces are sampled, with different probability. In this way we can react to changes in the network without using probe packets to test the condition of different routes.

5.5 Evaluation

To evaluate the performance of the RAAQM congestion control we simulate different network scenarios. All the FIBs in each node are manually pre-computed, simulating a routing protocol that computes routes to all the available sources. We start to analyze the protocol on a single-path scenario, to evaluate the stability region of the controller and the fairness in case of multiple flows. Then we simulate a multipath scenario to see how RAAQM, in addition to the forwarding strategy described by the algorithm in Figure 5.5, uses the available bandwidth in the network.

5.5.1 Single Path Simulation

In this simulation we consider the case where a single user is connected directly to the source of the content. The link capacity is set to 100Mbps. In this setting we start to evaluate the stability region of the controller, that depends on the two parameters Δp and β [13]. The results of the analysis are presented in Table 5.1.

Parameters		Infinite Buffer		Buffer = 100 pkts	
Δp	β	$\widetilde{Q}$	$\widetilde{W}$	Packet Loss (%)	$\widetilde{W}$
0.5	0.5	21.67	2.36	0	2.36
0.2	0.5	36.93	4.07	0	4.07
0.2	0.2	60.73	6.45	0	6.45
0.1	0.5	54.74	6.11	0	6.11
0.1	0.2	86.97	9.34	0.09	9.32
0.1	0.05	117.84	18.37	2.28	13.34
0.1	0.02	267.90	27.55	4.97	16.96
0.05	0.02	∞	∞	8.05	21.76

Table 5.1. Single path scenario: sensitivity analysis

In this simulation the user requires 10 different contents at the same time. We run two sets of simulations. In the first set we use a buffer, situated between the provider and the bottleneck link, with a really large size, indicated in the table as Infinite Buffer, to see the behavior of the controller in absence of packet loss. In the second set of simulations the size of the buffer is limited to 100 packets. In each simulation we change the values of Δp and β. In case of infinite buffer, we report the average queue size and the average cogestion window size, indicated respectively with $\widetilde{Q}$ and $\widetilde{W}$. In case of finite buffer a larger queue translates in packet losses, so, instead of $\widetilde{Q}$ we measure the percentage of packet loss.

As a general result we can say that for $\beta\Delta p > 2 \times 10^{-3}$, which is the region above the last line (indicated with an horizontal line), the controller is stable. In fact in the case $\Delta p = 0.05$ $\beta = 0.02$ the queue and the window sizes keep growing during the simulation and they never stabilize. Generally speaking the size of the queue at the bottleneck increases as the product $\beta\Delta p$ decreases. In the case of finite buffer, the instability of the controller translates in an higher loss percentage, that reduces the performances of the controller, increasing the transfer time of the content. In the rest of our simulations we set $\Delta p = 0.2$ and $\beta = 0.2$.

Figure 5.6 shows the instantaneous throughput of three different flows that share the same link capacity. As in the previous simulation we use a single link with 100Mbps capacity. The user issues three different requests, at 1 second, 5 seconds and 10 second after the beginning of the simulation, respectively. The plot shows that the link capacity is always fully used, in fact the sum of all the rates at each time is equal to 100Mbps. It also shows that, as soon as a new flow comes, the bandwidth is fairly shared among the active flows.

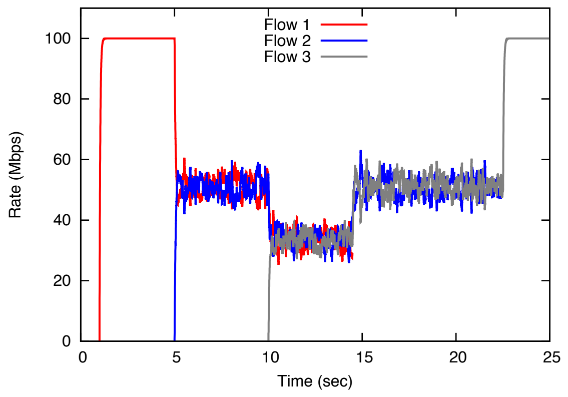

Figure 5.6. Single path scenario: fairness among flows

In this simulation we also measure the evolution of the size of the queue at the bottleneck and the size of the congestion window for each flow. The data are presented in Figure 5.7. The top part of the chart shows the size of the queue at the bottleneck in packets, while the lower part shows the window size for each flow, again in packets. The queue size reflects the number of flows in progress. This is due to the fact that RAAQM controls the difference between R_{min} and

R_{max}, and R_{min} increases every time that a new flow starts. As long as R_{min} is smaller than the RTT associate with a full buffer, the queue at the bottleneck grows with the number of flows. However, when we get close to the buffer saturation, the difference between R_{min} and R_{max} decreases, and this increases the probability to reduce the size of the window, as describe in Equation 5.1. In this way we can avoid congestion. The efficiency of the RAAQM controller is also guaranted by the fact that the queue at the bottleneck is never empty and so the link alway works at full rate.

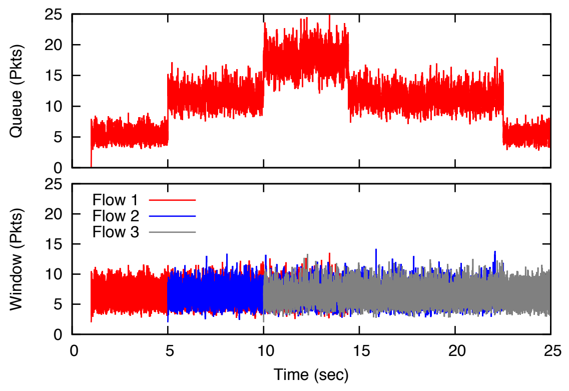

Figure 5.7. Single path scenario: queue and window evolution

5.5.2 Multipath Simulation

In this part of the evaluation we want to analyze the proposed RAAQM controller in addition to the forwarding strategy, using the multipath scenario presented in Figure 5.8. On the right side of the network there are 4 producer nodes, namely node i, j, k and l, and each producer stores the entire content catalog. The user is on the left side of the network, at node a. The capacity of each link is reported in the picture. The user, as well as all the producers, is connected to the network with an high capacity link (1Gbps) to avoid bottlenecks at the edge of the network that would prevent the usage of multiple paths. The figure also highlights the bottleneck with thicker lines: link (b, d) (30Mbps), link (c, e) (5Mbps) and link (c, f) (55Mbps).

We start with the evaluation of the forwarding strategy presented in the algorithm in Figure 5.5. Figure 5.9 shows the evolution of the split ratios, so the percentage of the traffic that goes on a certain route, calculated at node b, c and d during 5 second of simulation. The plain lines in the plot represent the

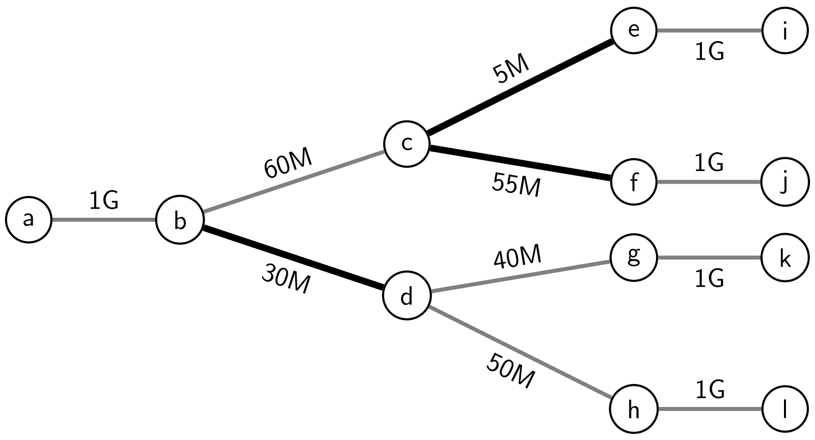

Figure 5.8. Multipath scenario topology

optimal percentage of traffic, computed manually, while the dashed lines represent the measured values. We show only 3 values, the others are obviously the complement of these values.

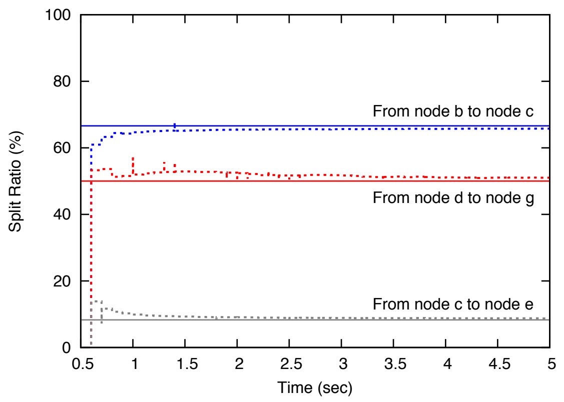

Figure 5.9. Multipath scenario: split ratio evolution

The optimal split ratio are easy to compute using the link capacities, and here we give some examples. The maximum rate achievable at node b is 90Mbps, which is the sum of the capacity of the links (b,c) and (b,d). Out of this 90Mbps we can send 60Mbps to node c and 30Mbps to node d. It is worth noticing that node d represents a particular case where the bottleneck is downstream with respect to the node. In this case the node can not saturate the bandwidth available upstream, so the load is divided equally over the two links (d,g) and (d,h). The final optimal split ratios are: at node b 66.6% of the traffic goes to node c and the rest to node d; on node c 8.3% of the traffic goes to node e and the rest to f; on node d the traffic splits evenly among g and h.

It is clear from Figure 5.5 that the simulation converges quickly to the optimal values. This simulation shows that our algorithm can dynamically compute the optimal weights [14].

In the last simulation we test our RAAQM algorithm in combination with the forwarding strategy on the topology in Figure 5.8. The results are presented in Figure 5.10 and 5.11.

In Figure 5.10 we show that a good forwarding strategy is necessary in order to exploit the entire bandwidth available in the network. We compare the throughput achieved with our forwarding strategy, in red, and the throughput in case of a random forwarding strategy, the blue line. In case of the random strategy, each node uniformly select an output interface, so the traffic is evenly distributed among the neighbors of the router. The blue line, labeled with *tot Random*, reports the total bandwidth used by the flows measured on the link between the user (node a) and node b. The average value is 20Mbps, that means that, for each route, we use only 5Mbps, which corresponds to the link with the smallest capacity, namely link $c - e$. Using the proposed forwarding strategy instead, we can fully utilize each bottleneck in the network (see labels in the picture), and the throughput measured between the user at node a and node b, labeled with *tot Fwd Strategy*, is really close to 90Mbps, that is the total capacity of our network.

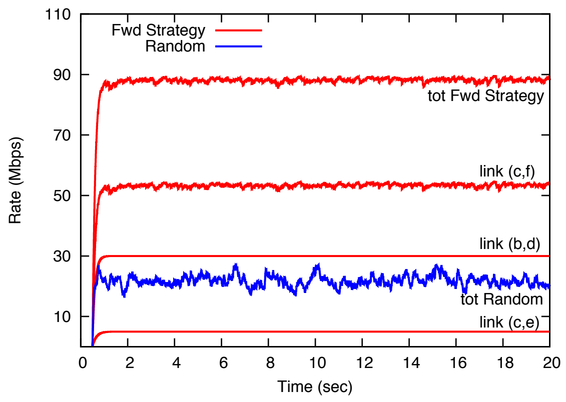

Figure 5.10. Multipath Scenario: Rate at Bottlenecks Links

Figure 5.11 shows the evolution of the queues, measured in packet, at the bottlenecks. In particular, the evolution of the queue on node e is presented in the top part of the picture, and the queue at node d is in bottom part of the picture. We do not show the queue at node f because it never grows over 15 packets. The maximum size of each queue is 100 packets. Both the charts show that the queues are quite stable and they never saturate. We drop only 0.007% of the packets, all of them in at node d, which is the most loaded one.

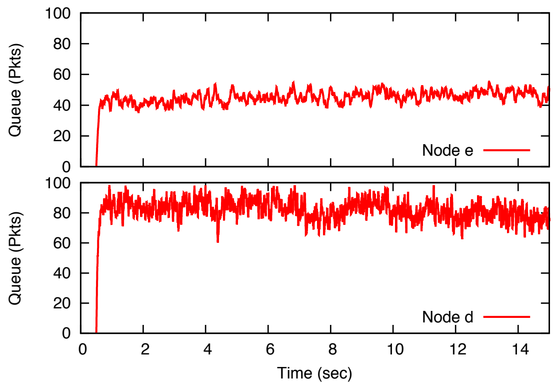

Figure 5.11. Multipath Scenario: Queues at Bottlenecks Links

In this section we evaluated RAAQM through simulations. However, this transport protocol is implemented as an extension of CCNx (the code is compatible with versions from 0.6.1 to 0.7.1). We used this implementation to run the experiments on the testbed Grid'5000.[1] In this evaluation we used different topologies: a topology similar to the one in Figure 5.8, a CDN like topology, the Abilene topology,[2] and a mobile back-haul topology [14]. RAAQM is used also as a transport protocol in other works based on CCN [62, 15].

[1]Grid'5000 webpage. https://www.grid5000.fr/

[2]Abilene topology. https://itservices.stanford.edu/service/network/internet2/abilene

Chapter 6

Conclusion and Future Work

The Internet changed dramatically since when it was created. The number of users that have access to the Internet surpassed 3 billions, and there will be about 50 billion objects connected by 2020.[1] The way we use the Internet also changed: from a network designed to share devices, to a content delivery network. User mobility also introduced many challenges that are not easily manageable in the current Internet. Information Centric Networking has the ambition to design a new Internet better suited for today's user needs and to solve all the problems arising from the tremendous growth of the network.

In this thesis we present a new ICN architecture called TagNet. We design and develop TagNet, and we conduct an extensive evaluation of our design. TagNet tries to give an answer to some of the fundamental problems that an ICN architecture should address. One of the goals is to better support the current and future uses of the Internet. TagNet does it by providing rich communications primitives, by allowing both push and pull communication at the network level. In fact, even if the majority of the Internet traffic can be handled with request/reply communication, many applications can benefit form a notification service at the network level, and many more may do the same in the future.

TagNet also empowers users and applications to address content with expressive addresses. These user-defined addresses are encoded in descriptors consisting of sets of tags. Descriptors can be used to describe some content, but they are also powerful enough to mimic the semantic of hierarchical names.

At a more engineering level, the implementation of TagNet allows scalability of the network as a whole, as a diverse decentralized internetwork, and fast packet forwarding and therefore high-throughput transmission. The combina-

[1]Connections Counter: The Internet of Everything in Motion. http://newsroom.cisco.com/feature-content?type=webcontent&articleId=1208342

tion of push and pull traffic is simple to implement, thanks to our routing scheme based on trees that also allows for good aggregation of routing information and therefore reduced sizes for the RIBs. This is a fundamental property to guarantee the scalability of the network. At the same time, the use of different, specialized addresses reduces the amount of in-network soft state, and allows for efficient packet forwarding and overall improved performance of the data plane.

The work done in this thesis explores many important aspects of the design and the development of a new network architecture. However, this can be considered just a stating point, since many aspects still need to be clarified, and many problems remain without an answer.

At the application level we still need to conduct an extensive comparative analysis of TagNet with other ICN proposals such as NDN and CCN, which are arguably the most developed and studied ICN architecture, and also the ones that are most similar to TagNet. Now we have all the building blocks for a complete prototype, such as the update algorithm and the forwarding engine, but we are still missing a complete implementation that can run on a real testbed. This is an important part for the future development of TagNet.

At the routing level we show that the architecture scales both in terms of memory requirements and in terms of updating time, also in case of significantly large RIBs. At this level there are still two main open questions: (i) how to build the trees and (ii) how to support user mobility.

To answer the first question we already achieved interesting results that are not part of this thesis. These results tell us that a few trees are actually enough to cover the Internet in a way that minimizes both latency and congestion [60]. However, in these studies, we do not take into account the routing polices that are so essential for a routing scheme at the AS level. BGP, in fact, allows each AS to choose different routes, mostly based on the business strategy of the operators rather than to minimize path lengths or other global topological metrics. In order to use our tree-based routing at the AS-level, we must provide a way for the ASes to create trees that, at least to some extent, satisfy their routing polices. In our current understanding of the routing problem in ICN, this goal poses nontrivial technical problems.

The other aspect that we simply touched upon in this dissertation is user mobility. One of the missions of ICN is to support mobility better than IP networks. TagNet seems to have a good way to address user mobility, thanks to the clear separation between content descriptors and locators, and therefore thanks to the combination of specialized forwarding algorithms. However, what we have designed so far is only a sketch of the mobility algorithm that we need to develop and study more extensively and in more detail.

The forwarding engine that we propose for TagNet also needs more investigation and refinement. Our evaluation shows that, thanks to the combination of content-based and locator-based forwarding, we can achieve a good throughput. However, despite the impressive improvement we obtained with respect to previous versions of our implementation, the pure content-based part is still not competitive if compared with IP forwarding in current core routers.

In an attempt to overcome these performance barriers, we also developed a GPU-based matcher that performs only the FAS algorithm, and that is capable of forwarding about 1 million packets per second with the 63M workload. The main problem with this algorithm, which is common to most GPU-based matching systems, is that it suffers from high latency, in particular in the order of hundreds of milliseconds. This level of latency makes this solution ineffective in real, reasonable-sized networks. In the future we plan to look also at other specific hardware, like TCAMs and FPGA. Still, the problem we pose (subset or "partial" matching) seems to be fundamentally more complex than the traditional forwarding problem in IP networks (prefix matching).

One aspect that we did not touch at all in the dissertation and requires more deep investigation is security. We did not conduct any security study on our architecture, but we are well aware that there are many issues related to security. Some of these issues are not particularly new or challenging for TagNet. For example, the problem of authenticating content is the same as in CCN, for which a number of architectural and technical solutions are available. Without exploring all the details, we can assume that the content can be digitally signed by the publisher, and then verified by the end-user application or even by the network on the user's behalf. This is certainly true for pull communication, since request and reply packets have the same semantic of interests and data packet in CCN. However, we think that the same technique can also be applied to notification packets, where the final user can verify the authenticity of the packet.

Another important security problem is the risk of DDoS attacks that may exploit some aspect of the architecture. In fact, it is easy to flood the network with useless messages. For example, a malicious user may send a series of request or notification packets with many tags in the descriptor. In this way , as we already discuss in the second Chapter, there is an high probability that the packets match many entries in the FIBs of the routers, and so, the packets get forwarded everywhere. We envision different ways to prevent this type of attack. One way is to charge the users according to the expected traffic that they will generated in the network. A packet with more tags may cost more, simply because more tags mean a higher probability to match predicates, and therefore a higher network cost for delivering the packet. Another technical solution to the problem is to implement some kind of firewall in the routers. A router may not accept messages with a descriptor that has a Hamming weight that is too high, or, when some packets have a fan out that is higher than a certain threshold, the router may decide to discard the packet, or forward it to a subset of the matching interfaces. Both these strategies are easy to implement also in our forwarding engine and they are useful to prevent attacks.

Bibliography

[1] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A survey of information-centric networking. *Communications Magazine, IEEE*, 50(7):26–36, 2012.

[2] B. Ahlgren et al. Second netinf architecture description. *4WARD EU FP7 Project, Deliverable D-6.2 v2.0*, 2010.

[3] R. Ahmed, M.F. Bari, S.R. Chowdhury, M.G. Rabbani, R. Boutaba, and B. Mathieu. aroute: A name based routing scheme for information centric networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 90–94, April 2013.

[4] M. Amadeo, C. Campolo, A. Iera, and A. Molinaro. Named data networking for iot: An architectural perspective. In *Networks and Communications (EuCNC), 2014 European Conference on*, pages 1–5, June 2014.

[5] D. Ardelean, E. Blanton, and M. Martynov. Remote active queue management. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '08, pages 21–26, New York, NY, USA, 2008. ACM.

[6] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *the Journal of machine Learning research*, 3:993–1022, 2003.

[7] A. Borodin, R. Ostrovsky, and Y. Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, pages 312–321, New York, NY, USA, 1999. ACM.

[8] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[9] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron. Virtual ring routing: Network routing inspired by dhts. *SIGCOMM Comput. Commun. Rev.*, 36(4):351–362, August 2006.

[10] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, I. Stoica, and S. Shenker. Rofl: Routing on flat labels. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 363–374, New York, NY, USA, 2006. ACM.

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

[12] G Carofiglio, M. Gallo, and L. Muscariello. Joint hop-by-hop and receiver-driven interest control protocol for content-centric networks. In *Proceedings of the second edition of the ICN workshop on Information-centric networking*, pages 37–42. ACM, 2012.

[13] G. Carofiglio, M. Gallo, L. Muscariello, and M Papalini. Multipath congestion control in content-centric networks. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, 2013.

[14] G. Carofiglio, M. Gallo, L. Muscariello, M. Papalini, and S. Wang. Optimal multipath congestion control and request forwarding in information-centric networks. In *International Conference on Network Protocol (ICNP)*, 2013.

[15] G. Carofiglio, M. Gallo, L. Muscariello, and D. Perino. Scalable mobile backhauling via information-centric networking. In *Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on*, pages 1–6, April 2015.

[16] A. Carzaniga, K. Khazaei, M. Papalini, and A.L. Wolf. Is information-centric multi-tree routing feasible? In *Proceedings of the ACM SIGCOMM Workshop on Information-Centric Networking*, ICN '13, 2013.

[17] A. Carzaniga, M. Papalini, and A. L. Wolf. Content-based publish/subscribe networking and information-centric networking. In *Proceedings of the ACM SIGCOMM Workshop on Information-Centric Networking*, ICN '11, pages 56–61, 2011.

[18] A. Carzaniga, M.J. Rutherford, and A.L. Wolf. A routing scheme for content-based networking. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 2, pages 918–928 vol.2, 2004.

[19] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in Lecture Notes in Computer Science, pages 59–68. Springer-Verlag, October 2001.

[20] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 163–174, 2003.

[21] M. Charikar, P. Indyk, and R. Panigrahy. New algorithms for subset query, partial match, orthogonal range searching, and related problems. In *Automata, Languages and Programming*, pages 451–462. Springer, 2002.

[22] J. Chen, M. Arumaithurai, X. Fu, and K.K. Ramakrishnan. Coexist: A hybrid approach for content oriented publish/subscribe systems. In *Proceedings of the Second Edition of the ICN Workshop on Information-centric Networking*, ICN '12, pages 31–36, New York, NY, USA, 2012. ACM.

[23] J. Chen, M. Arumaithurai, L. Jiao, X. Fu, and KK Ramakrishnan. Copss: An efficient content oriented publish/subscribe system. In *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, pages 99–110. IEEE, 2011.

[24] R. Chiocchetti, D. Perino, G. Carofiglio, D. Rossi, and G. Rossini. Inform: a dynamic interest forwarding mechanism for information centric networking. In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*, ICN '13, pages 9–14, New York, NY, USA, 2013. ACM.

[25] D. D. Clark, M. L. Lambert, and L. Zhang. Netblt: A high throughput transport protocol. *SIGCOMM Comput. Commun. Rev.*, 17(5):353–359, August 1987.

[26] H. Dai, J. Lu, Y. Wang, and B. Liu. A two-layer intra-domain routing scheme for named data networking. In *Global Communications Conference (GLOBECOM), 2012 IEEE*, pages 2815–2820, Dec 2012.

[27] C. Dannewitz, J. Golic, B. Ohlman, and B. Ahlgren. Secure naming for a network of information. In *INFOCOM IEEE Conference on Computer Communications Workshops , 2010*, pages 1–6, March 2010.

[28] P. Th. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, pages 114–131, 2003.

[29] S. Eum, K. Nakauchi, M. Murata, Y. Shoji, and N. Nishinaga. Catt: potential based routing with content caching for icn. In *Proceedings of the second edition of the ICN workshop on Information-centric networking*, ICN '12, pages 49–54, New York, NY, USA, 2012. ACM.

[30] W. Fenner, M. Rabinovich, K.K. Ramakrishnan, D. Srivastava, and Yin Zhang. Xtreenet: scalable overlay networks for xml content dissemination and querying (synopsis). In *Web Content Caching and Distribution, 2005. WCW 2005. 10th International Workshop on*, pages 41–46, 2005.

[31] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 43–56, New York, NY, USA, 2000. ACM.

[32] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, August 1993.

[33] J. Francois, T. Cholez, and T. Engel. Ccn traffic optimization for iot. In *Network of the Future (NOF), 2013 Fourth International Conference on the*, pages 1–5, Oct 2013.

[34] Z. Gao, A. Venkataramani, J. F. Kurose, and S. Heimlicher. Towards a quantitative comparison of location-independent network architectures. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 259–270, New York, NY, USA, 2014. ACM.

[35] J.J. Garcia-Luna-Aceves. Name-based content routing in information centric networks using distance information. In *Proceedings of the 1st International Conference on Information-centric Networking*, INC '14, pages 7–16, New York, NY, USA, 2014. ACM.

[36] J.J. Garcia-Luna-Aceves. Routing to multi-instantiated destinations: Principles and applications. In *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, pages 155–166, Oct 2014.

[37] J.J. Garcia-Luna-Aceves and M. Mirzazad-Barijough. Enabling correct interest forwarding and retransmissions in a content centric network. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 135–146, Washington, DC, USA, 2015. IEEE Computer Society.

[38] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox. Information-centric networking: Seeing the forest for the trees. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 1:1–1:6, 2011.

[39] A. Goel and P. Gupta. Small subset queries and bloom filters using ternary associative memories, with applications. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 143–154. ACM, 2010.

[40] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Commun. ACM*, 42(2):39–41, February 1999.

[41] M. Gritter and D. R. Cheriton. An architecture for content routing support in the internet. In *Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems - Volume 3*, USITS'01, pages 4–4, Berkeley, CA, USA, 2001. USENIX Association.

[42] P. Gupta and N. McKeown. Algorithms for packet classification. *Network, IEEE*, 15(2):24–32, 2001.

[43] H. Han, S. Shakkottai, and et al. Hollot. Multi-path tcp: a joint congestion control and routing scheme to exploit path diversity in the internet. *IEEE/ACM Trans. Netw.*, 14(6):1260–1271, December 2006.

[44] M. Honda, Y. Nishida, L. Eggert, P. Sarolahti, and H. Tokuda. Multipath congestion control for shared bottleneck. In *Proc. PFLDNeT workshop*, pages 19–24, 2009.

[45] A. K. M. M. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang. Nlsr: named-data link state routing protocol. In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*, ICN '13, pages 15–20, New York, NY, USA, 2013. ACM.

[46] H. Hsieh, K.n Kim, Y. Zhu, and R. Sivakumar. A receiver-centric transport protocol for mobile hosts with heterogeneous wireless interfaces. In

Proceedings of the 9th Annual International Conference on Mobile Computing and Networking, MobiCom '03, pages 1–15, New York, NY, USA, 2003. ACM.

[47] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, CoNEXT '09, pages 1–12, 2009.

[48] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander. LIPSIN: Line speed publish/subscribe inter-networking. In *Proceedings of the ACM SIGCOMM Conference on Data Communication*, 2009.

[49] A.W. Kazi and H. Badr. Pit and cache dynamics in ccn. In *Global Communications Conference (GLOBECOM), 2013 IEEE*, pages 2120–2125, Dec 2013.

[50] F. Kelly and T. Voice. Stability of end-to-end algorithms for joint routing and rate control. *SIGCOMM CCR*, 35(2):5–12, 2005.

[51] M. Koponen, T.and Chawla, B. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. *SIGCOMM Comput. Commun. Rev.*, 37(4):181–192, 2007.

[52] A. Kuzmanovic and E. W. Knightly. Receiver-centric congestion control with a misbehaving receiver: Vulnerabilities and end-point solutions. *Elsevier Computer Networks*, 2007.

[53] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[54] P. Mahadevan, E. Uzun, S. Sevilla, and J.J. Garcia-Luna-Aceves. Ccn-krs: A key resolution service for ccn. In *Proceedings of the 1st International Conference on Information-centric Networking*, ICN '14, pages 97–106, New York, NY, USA, 2014. ACM.

[55] I. Moiseenko, M. Stapp, and D. Oran. Communication patterns for web interaction in named data networking. In *Proceedings of the 1st International Conference on Information-centric Networking*, ICN '14, pages 87–96, New York, NY, USA, 2014. ACM.

[56] D.R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968.

[57] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M.o Munafò, K. Papagiannaki, and P. Steenkiste. The cost of the "s" in https. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 133–140, New York, NY, USA, 2014. ACM.

[58] K. Nichols and V. Jacobson. Controlling queue delay. *Queue*, 10(5):20:20–20:34, May 2012.

[59] F. Papadopoulos, D. Krioukov, M. Boguñá, and A. Vahdat. Greedy forwarding in dynamic scale-free networks embedded in hyperbolic metric spaces. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 2973–2981, Piscataway, NJ, USA, 2010. IEEE Press.

[60] M. Papalini, A. Carzaniga, K. Khazaei, and A. L. Wolf. Scalable routing for tag-based information-centric networking. In *Proceedings of the 1st International Conference on Information-centric Networking*, ICN '14, pages 17–26, New York, NY, USA, 2014. ACM.

[61] M. Papalini, K. Khazaei, A. Carzaniga, and A. L. Wolf. Scalable routing for tag-based information-centric networking. Technical Report 2014/01, University of Lugano, February 2014.

[62] D. Perino, M. Gallo, R. Boislaigue, L. Linguaglossa, M. Varvello, G. Carofiglio, L. Muscariello, and Z. Ben Houidi. A high speed information-centric network in a mobile backhaul setting. In *Proceedings of the 1st International Conference on Information-centric Networking*, ICN '14, pages 199–200, New York, NY, USA, 2014. ACM.

[63] D. Perino, M. Varvello, L. Linguaglossa, R.l Laufer, and R. Boislaigue. Caesar: A content router for high-speed forwarding on content names. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 137–148, New York, NY, USA, 2014. ACM.

[64] C. E. Perkins and D. B. Johnson. Route optimization for mobile ip. *Cluster Computing*, 1(2):161–176, 1998.

[65] H. Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the 40th annual ACM symposium on Theory of computing (STOC'08)*, 2008.

[66] V. C. Ravikumar and R. N. Mahapatra. Tcam architecture for ip lookup using prefix properties. *Micro, IEEE*, 24(2):60–69, 2004.

[67] D. Raychaudhuri, K. Nagaraja, and A. Venkataramani. Mobilityfirst: A robust and trustworthy mobility-centric architecture for the future internet. *SIGMOBILE Mob. Comput. Commun. Rev.*, 16(3):2–13, December 2012.

[68] Y. Ren, J. Li, S. Shi, L. Li, and X. Chang. An interest control protocol for named data networking based on explicit feedback. In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, pages 199–200, May 2015.

[69] R. L Rivest. Partial-match retrieval algorithms. *SIAM Journal on Computing*, 5(1):19–50, 1976.

[70] L. Saino, C. Cocora, and G. Pavlou. Cctcp: A scalable receiver-driven congestion control protocol for content centric networking. In *IEEE ICC*, 2013.

[71] L. Saino, I. Psaras, and G. Pavlou. Hash-routing schemes for information centric networking. In *Proceedings of the 3rd ACM SIGCOMM Workshop on Information-centric Networking*, ICN '13, pages 27–32, New York, NY, USA, 2013. ACM.

[72] P. Sinha, T. Nandagopal, N. Venkitaraman, R. Sivakumar, and V. Bharghavan. Wtcp: A reliable transport protocol for wireless wide-area networks. *Wirel. Netw.*, 8(2/3):301–316, March 2002.

[73] W. So, A. Narayanan, and D. Oran. Named data networking on a router: Fast and dos-resistant forwarding with hash tables. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, pages 215–226, Piscataway, NJ, USA, 2013. IEEE Press.

[74] N. Spring, R. Mahajan, and T. Wetherall, D.and Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, February 2004.

[75] I. Stoica, D. Adkins, S. Zhuang, and S. Shenker, S.and Surana. Internet indirection infrastructure. *IEEE/ACM Trans. Netw.*, 12(2):205–218, April 2004.

[76] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '01, pages 1–10, New York, NY, USA, 2001. ACM.

[77] D. Trossen et al. Pursuit, publish subscribe internet technology: Architecture definition, components descriptions and requirements. *PURSUIT EU FP7 Project, Deliverable D2.3*, 2011.

[78] V. Tsaoussidis and C. Zhang. Tcp-real: Receiver-oriented congestion control. *Comput. Netw.*, 40(4):477–497, November 2002.

[79] M. Virgilio, G. Marchetto, and R. Sisto. Pit overload analysis in content centric networks. In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*, ICN '13, pages 67–72, New York, NY, USA, 2013. ACM.

[80] Y. Wang, N. Rozhnova, A. Narayanan, D. Oran, and I. Rhee. An improved hop-by-hop interest shaper for congestion control in named data networking. In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*, pages 55–60. ACM, 2013.

[81] Y. Wang, B. Xu, D. Tai, J. Lu, T. Zhang, H. Dai, B. Zhang, and B. Liu. Fast name lookup for named data networking. In *Quality of Service (IWQoS), 2014 IEEE 22nd International Symposium of*, pages 198–207, May 2014.

[82] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang. Wire speed name lookup: a gpu-based approach. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 199–212, Berkeley, CA, USA, 2013. USENIX Association.

[83] R. Wetzker, C. Zimmermann, and C. Bauckhage. Analyzing social bookmarking systems: A del. icio. us cookbook. In *Proceedings of the ECAI 2008 Mining Social Data Workshop*, pages 26–30, 2008.

[84] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proc. of NSDI'11*, 2011.

[85] G. Xylomenos, C.N. Ververidis, V.A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K.V. Katsaros, and G.C. Polyzos. A survey of information-centric networking research. *Communications Surveys Tutorials, IEEE*, 16(2):1024–1049, Second 2014.

[86] C. Yi, J. Abraham, A.r Afanasyev, L. Wang, Beichuan Zhang, and L. Zhang. On the role of routing in named data networking. In *Proceedings of the 1st International Conference on Information-centric Networking*, ICN '14, pages 27–36, New York, NY, USA, 2014. ACM.

[87] C. Yi, A. Afanasyev, L. Wang, B. Zhang, and L. Zhang. Adaptive forwarding in named data networking. *SIGCOMM Comput. Commun. Rev.*, pages 62–67, 2012.

[88] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proc. of Usenix ATEC*, 2004.

[89] Y. Zhang, H. Zhang, and L. Zhang. Kite: A mobility support scheme for ndn. In *Proceedings of the 1st International Conference on Information-centric Networking*, ICN '14, pages 179–180, New York, NY, USA, 2014. ACM.

[90] K. Zheng, C. Hu, H. Liu, and B. Liu. An ultra high throughput and power efficient tcam-based ip lookup engine. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1984–1994. IEEE, 2004.