POLITECNICO DI TORINO

SCUOLA DI DOTTORATO

PhD Course in Computer and Control Engineering – XXVIII cycle

PhD Dissertation

Study and analysis of innovative network protocols and architectures

Matteo Virgilio
student id: 199850

Tutor	**Course Coordinator**
dr. Guido Marchetto	prof. Matteo Sonza Reorda

May 2016

† A mia nonna Sara

Summary

In the last years, Internet has grown at an unbelievable rate, pervading almost all the aspects connected to our everyday life. Work, entertainment, social networking, mobility, long-distance interactions: these are all examples of contexts that have been extensively touched and often revolutionized by the use of the ICT technologies. Nevertheless this trend is expected to become even more accentuated in the next future thanks to the introduction of different new paradigms: Internet Of Things, Virtual/Augmented Reality, Cloud/Fog Computing just to mention few. One important key point to understand is that the underlying technologies and the vast community of researchers/developers that revolves around it, has the important task of constantly bringing innovation to support modern IT applications and also to make them sustainable from different viewpoints: environmental, cultural, social, political. Most of the challenges for the next years are heavily related to the different directions that the ICT area is going to pursue and we, as a community of "thinkers", have to carefully assess the benefits that a certain paradigm shift can bring to the world as opposed to the drawbacks that a particular technology can contain in its definition or in the way it can be (mis)used by the final users.

In this respect, among all the challenges that are continuously raising new questions, we will focus on the security and safety aspects that some new paradigms, currently proposed in the networking panorama, are posing as open questions. This task is not trivial since, when we talk about security (in its widest meaning), we also need to talk about the "side thinking" approach which basically requires to take into consideration all the possible scenarios in which a given paradigm or technology can operate and the ability to foresee possible malicious behaviors which are not the expected (or desired) ones. As a consequence of this fact, it is possible, in general, to study security issues from different standpoints. Some people are more comfortable in analyzing new protocols or system architectures by reasoning on their formal definition, thus defining "rules" that the system under analysis should obey to. Others are more prone to detect different types of errors or design flaws by looking directly at the system itself. In this case, we have a number of different methodologies that have been widely used since the born of the first computer program: black box testing, white box testing, static code analysis and so on. If the system under test is a network protocol, simulation is an appealing approach that actually worked and

still works nowadays to assess the behavior in the most representative operating scenarios. Please notice that all these techniques should not be seen as mutually exclusive but rather as complementary ways to ensure an higher level of confidence for the achieved results where, most of the time, a desirable result is "the system behaves as expected in all situations" (clearly formalized in some sane way).

In this thesis, we will analyze the security implications of different newly proposed network architectures and we will leverage a variety of techniques based on the evolving context of the document. This will give us the opportunity to introduce in depth all the topics we have covered and the different approaches that we have successfully applied to all of them. Since the focus is especially placed on innovative protocols and architectures, we specifically focused on two dominant proposals that we currently have in the networking arena, namely the Content-Centric Networking (CCN) and the Software Defined Networking (SDN). Even if they target rather orthogonal aspects (thus bringing innovation in divergent directions), they can be seen as vivid expressions of the trends the community is willing to pursue in the next future; while the CCN mission is to support and optimize content dissemination by completely change the way information are actually routed and delivered in the Internet, SDN aims at making the network more programmable and, thus, flexible.

More specifically, we developed a full fledged Java simulator to represent various operating scenarios of the CCN protocol in order to assess mechanisms and security implications that we have found interesting and worth deepening from a security standpoint. Results presented in Section 1.5 are obtained by means of our simulation tool while results presented in the following sections are obtained using an extended version of ndnSIM, a reference simulator available in the CCN community, which was not yet released when we started our work and that we extensively customized to simulate our network scenarios in conjunction with our proposed mechanisms aimed at extending the features of the CCN architecture. For what concerns Section 2.1.1, our main contributions are in the formal verification activity, where a complete model of the proposed algorithm for the data exchange management in NFV contest and a set of desired properties have been developed to ensure system safety and consistency. In Section 2.2.1 our main contribution is in the network functions modeling activity, where we extended the presented verification tool to also support a wider category of stateful network functions, that we considered relevant in complex SDN/NFV deployment scenarios.

Contents

List of Tables

List of Figures

Chapter 1

CCN: an architecture for the future Internet

Portions of this chapter were previously published in [84, 86, 87]

When the Internet was born, it was not so easy to foresee all the possible future use cases, either for the users and also (especially) for the designers. As a matter of fact, most of the principles introduced in the TCP/IP stack were mainly inherited from the traditional POTS[1] network especially the part related to the identification of an endpoint by means of a location dependent identifier (what is commonly known as IP address) and also the conversational approach based on two hosts that exchange packets with each other. This can be easily observed by simply looking at the network layer packet structure, which contains a source and a destination field, thus implementing this idea of couples of peers that communicate thanks to an invisible pipe transporting bits from one place to another. Clearly, there exist a lot of differences in terms of requirements between the TCP/IP stack operating mode and the traditional circuit switching networks, not to mention the different level/type of services they export to their utilizers. Anyway, it is undoubtedly true that some concepts belonging to the telephony environment were inherited and wisely adopted also in the TCP/IP protocol suite.

The conversational approach was perfectly suitable for many use cases envisioned at design time (and also nowadays). For example, a very common scenario widely spread since the beginning of the Internet revolution, was to have a single smart and powerful computer (e.g. mainframe/server) closed into a room with sufficient space and air, and a plethora of simple devices connecting to it to share and use resources, intended here as a set of computing modules or as a set of blocks that store a massive amount of data. It is easy to understand that a given user was interested in accessing

[1] Plain Old Telephone Service

a *specific* mainframe. For example, a researcher could be interested in accessing his department mainframe in order to perform some simulations on his last model and get the results back after days of work. Clearly, in a similar scenario, where the user wants to reach a specific location (the department mainframe) with some specific access policies, there is nothing wrong with the conversational paradigm and the approach based on couples of peers exchanging packets works definitely fine. Still, a VoIP call is another example of communication entirely based on data transfer between two very specific hosts. However, if we consider the evolution of the Internet since these use cases, we have to admit at least two points:

1. the devices that were classified as simple I/O end terminals are now becoming smarter and smarter resembling very much the server they are connecting to (at least in terms of potential);

2. the mainframe/simple client use case has enormously evolved especially because the users of the Internet are no more highly skilled and highly focused researchers but more and more casual users, practically the entire world population, and they are of course not highly focused on IT technologies and their main purpose is to access the popular data they are interested in (sports, celebrities, news, social networking, . . .).

Considering this evolution, some researchers in the world started to think that the network, as we conceive it today, is not so optimized for the use we make of it. Most of the time, users do not care much about the location of a given information but they are actually interested in retrieving the content itself, no matter where it is located. Still, the information they are looking for is likely interesting for many others and the dissemination of it could (should) be aided by the network. Unfortunately this is not the case in the TCP/IP world albeit additional mechanism have been put in play to cope exactly with this problem during the years (through CDNs, web caches and so on). Just to be more concrete, let us think about the latest music video from one famous international pop singer. In this quite generic scenario (where generic here means "very common"), it is very likely that many users of the network will transfer the video from the original source (a media server, a YouTube like service, . . .) to their final devices (smartphones, laptops). This will result in a massive amount of logical point-to-point links between each user and the server which ends up in being massively overwhelmed by a huge amount of identical requests. A side effect of this operating mode is the network congestion, especially in proximity of the content providers which must overdimension their links in order to support the traffic peak, and also a degradation in terms of quality of experience perceived by the end users. Essentially, the problem stays in the fact that the famous music video is transferred over the same wires millions of times (or billions, if we look at the statistics of many popular videos on YouTube) and this clearly leads to an under-optimized use of the network resources, which are repeatedly used to transfer the same bits. Figure 1.1 shows a graphical representation of a network congestion due to the content provider bottleneck.

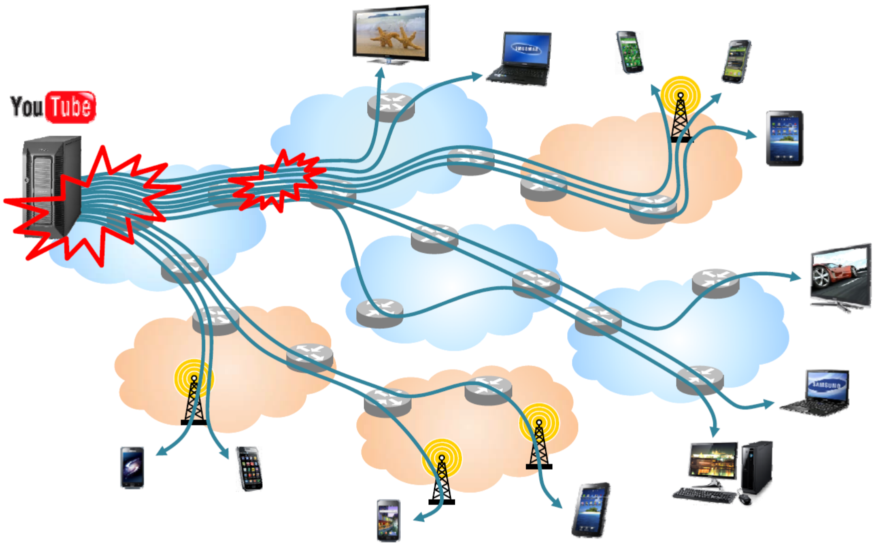


Figure 1.1: Network congestion in traditional networks

Starting from all these valid considerations, we assisted in the last years to the creation of a movement which targets the design and the implementation of a new network architecture for the future Internet, that is an alternative approach to the routing problem that can meet the increasing demand for content distribution across the globe. This initiative is generically called Information Centric Networking (ICN) and incorporates a number of tentative architectures that have been proposed during the last years to tackle the complex problem of optimizing the content dissemination by means of a full fledged protocol suite. One of the most prominent proposals in this area is certainly the Content-Centric Networking (CCN) solution, which has been originally proposed by Van Jacobson in 2009[37]. The essential idea here is to deploy a cache in all the nodes installed in the network (edge devices, core routers and so on) in order to maintain a local repository where to store the most popular contents. Each user requesting a given content has an opportunity to retrieve the information from one of the different caches scattered around the network thus minimizing the content retrieval latency. In this renewed scenario, the network would appear less congested due to the optimization introduced by caches and the already depicted scenario (Figure 1.1) would change according to Figure 1.2. As it is evident from the picture, assuming a reasonable hit ratio for the caches, the network congestion can be significantly reduced and the dissemination of content would be greatly aided according to the CCN mechanism. In the following we will detail in depth different features of this solution and also some drawbacks that must be more

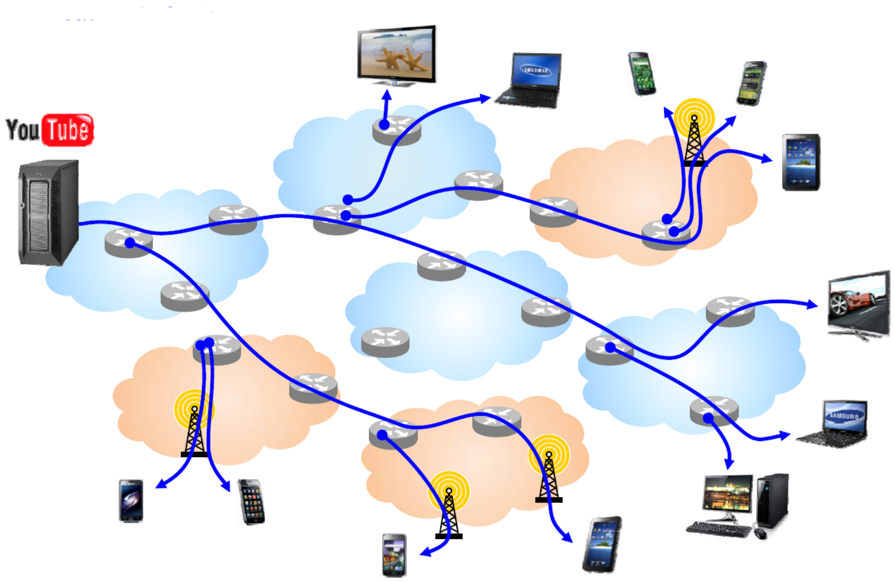


Figure 1.2: Network congestion improved in a CCN network

closely evaluated, especially because, to the best of our knowledge, no commercial implementation of CCN is yet available in any production network (even though some prototypes are currently under development in different research laboratories) hence no actual statistics can be gathered about the protocol performance in real conditions.

The aspects that we analyze in the first part of this thesis are related to security, scalability and performance in terms of generated traffic and delay between the content request and the content retrieval. We start our work with the analysis of a Distributed Denial of Service (DDoS) attack that has been introduced in CCN due to its stateful nature. We evaluate the problem by quantifying the impact on a real telecom operator scenario (by means of an ad-hoc simulator) and then we also analyze some possible countermeasures that have been implemented and adapted for our specific use case (the network of Telecom Italia, TI). After covering the DDoS security issue, we move towards the study of some scalability problems that can be encountered while implementing PUSH applications on top of CCN. Since the working mode of the protocol is inherently PULL, designing and implementing modern applications that work in PUSH mode appears to be rather challenging. We will propose an architecture to overcome some of the existing limitations with a special attention on the protocol scalability.

1.1 Introduction: modernizing the Internet

Starting from the valid claim that in a content distribution network it is actually important to address resources themselves, rather than their physical location, CCN aims to change the traditional network operation by making all network devices name-aware. Then, routing decisions are taken according to the name of the resource the user is requesting.

The approach is promising and may represent one of the most significant innovations in the networking field. However, its new operating mode opens the path for possible issues and threats that were not present in the traditional network. These have to be properly investigated in order to have an exhaustive evaluation of the overall CCN architecture. For example, content delivery is based on a status information (the requested URI) that has to be maintained at nodes in the so called Pending Interest Table (PIT). This may pose strict constraints in terms of reliability and scalability. In fact, this table may overflow, with consequent service disruption and possible network collapse. Furthermore, these constraints might be even exploited by attackers in order to slow down or even interrupt the normal network operation. In the next sections we deal with this specific problem by providing a performance evaluation of some possible PIT architectures in terms of resilience to (possibly malicious) overload conditions. In particular, we consider three PIT architectures, referenced in the available CCN literature:

- (*i*) a PIT storing all the bytes composing an URI, which we call SimplePIT;
- (*ii*) a PIT storing a fixed length entry for an URI, which we call HashedPIT;
- (*iii*) a PIT implemented through multiple Bloom Filters placed in each router interface, as described in [93]. This solution is known as DiPIT.

The experiments are conducted by means of an ad-hoc simulator, designed to recreate the behavior of a CCN network and to track memory usage at CCN nodes. In order to obtain significant results, the simulator implements the topology of a prominent Italian ISP.

1.2 Background

1.2.1 Names

CCN is built around the concept of named data. Each piece of content must have a unique name through which it can be retrieved by the different applications. For instance, "*/nytimes/website/index.html/0*" could be the first chunk name of the New York Times home page. In the same way, all the resources disseminated throughout the network must have analogous structured and hierarchical names. Notice that these names are not flat but aggregate on a name space basis so that a router can store a single entry in the routing table (e.g. “*/nytimes/**") to correctly forward all the requests addressed to a specific domain.

1.2.2 Packets

The mechanism provided to retrieve a content relies on a special packet called *Interest* which is used by clients to specify the name of the requested resources. Intermediate nodes route this packet towards the content custodian, by inspecting the name and matching it against the routing table.

Resources are then propagated by another kind of transport unit which is called *Data* packet and it is meant to carry the content payload. Some additional information are embedded within the Data packet in order to allow the producer to digitally sign its content and make its public key available to the world for the verification process. Figure 1.3 shows the main packet fields.

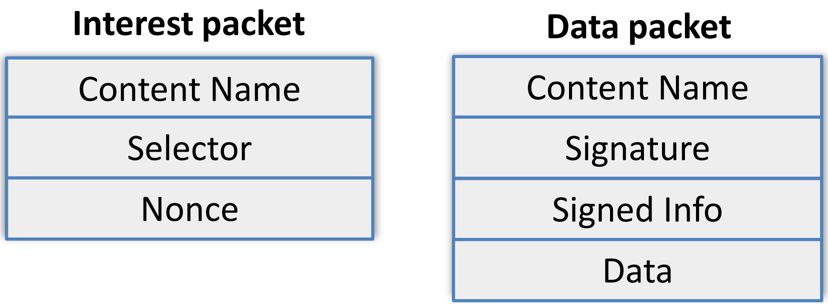


Figure 1.3: CCN packets[37]

1.2.3 Forwarding

The CCN routing engine is based on three internal data structures that are used to perform different tasks at the packet reception:

- *Content Store* (CS) is a local repository that acts as a node cache. All Data packets (or part of them, according to the cache replacement algorithm) may be stored in this local portion of the memory in order to serve future requests and speed up the response delivery;

- *Pending Interest Table* (PIT) is the data structure used to annotate all the forwarded Interest packets. Through this table, the node can “remember" from which interfaces it received a request for a given content hence it is able to correctly forward the response packet (when it will be generated);

- *Forwarding Information Base* (FIB) is the exact structure that we also have in traditional IP routers. Clearly, in the CCN architecture, the routing decision is taken according to the name present in the Interest packet so this table is used in combination with a Longest Prefix Match algorithm against the set of name components.

The way the two types of packets are processed by the CCN routing engine is asymmetric. In Figure 1.4, the Interest forwarding process is briefly summarized.

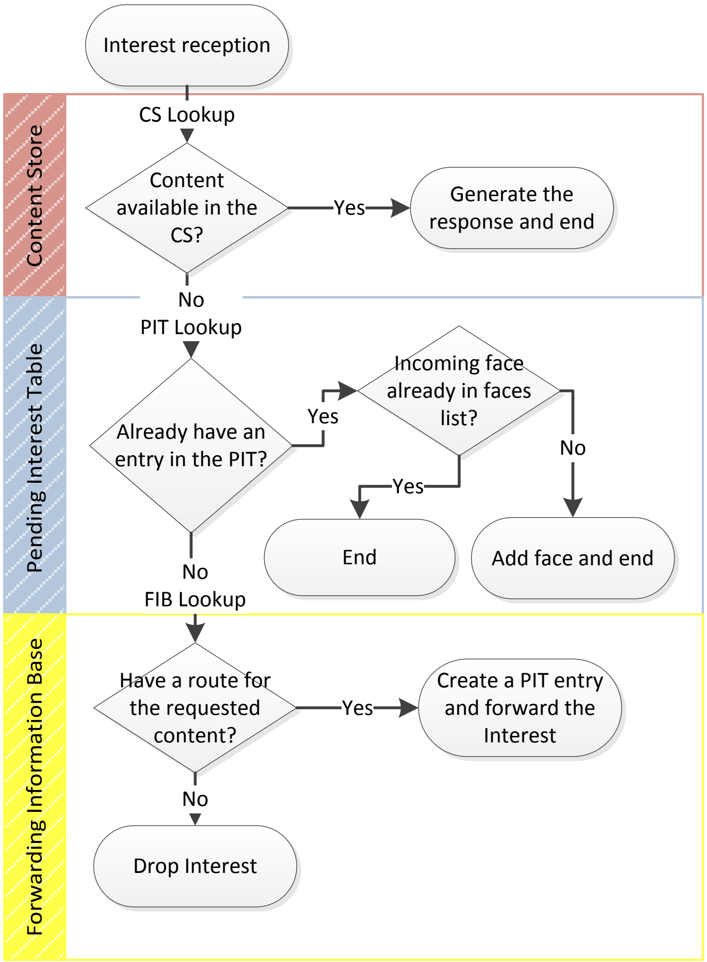

Interest reception
Content Store
CS Lookup
Content available in the CS?
Yes
Generate the response and end
No
Pending Interest Table
PIT Lookup
Already have an entry in the PIT?
Yes
Incoming face already in faces list?
Yes
No
End
Add face and end
No
FIB Lookup
Forwarding Information Base
Have a route for the requested content?
Yes
Create a PIT entry and forward the Interest
No
Drop Interest

Figure 1.4: Interest processing

Figure 1.5 shows how Data packets are not routed but they only follow the reverse path drawn by the Interests.

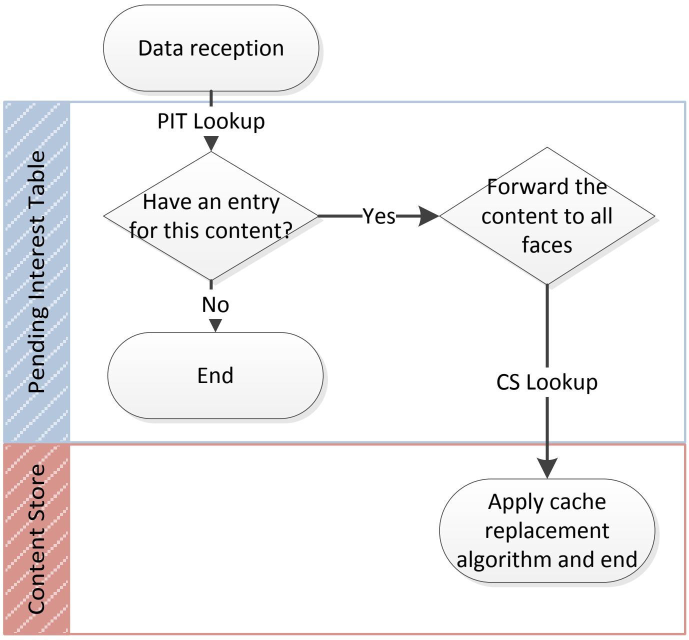
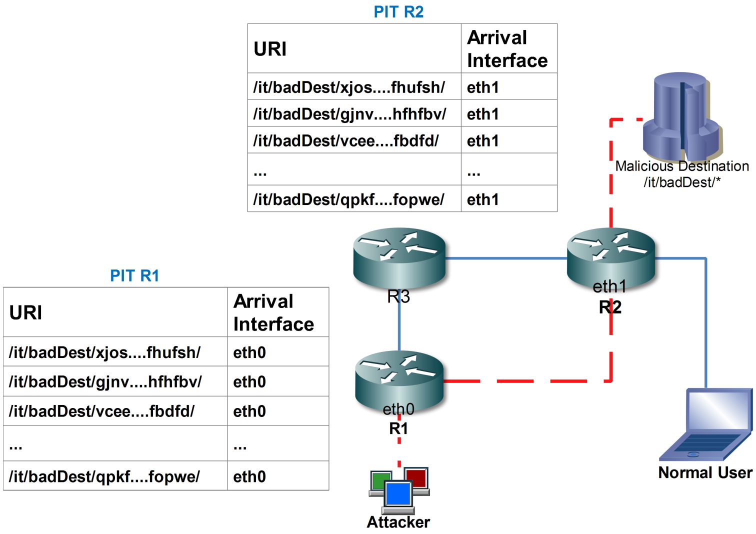

Data reception
PIT Lookup
Have an entry for this content?
Yes
Forward the content to all faces
No
End
CS Lookup
Apply cache replacement algorithm and end
Pending Interest Table
Content Store

Figure 1.5: Data processing

1.3 Security: problems and solutions

The PIT is the fundamental structure used to maintain the state of each active flow. It grows with users sending their Interests and shrinks when Data packets arrive at the router. Considering the access speed required for such a structure and the possibilities offered in this sense by current memory technologies, the PIT size might represent a bottleneck for the entire CCN infrastructure.
The problem might be exacerbated by a massive usage of long Interest LifeTimes, which would further increase the number of simultaneous entries in the PIT. Despite the *pull* nature of CCN, this possibility cannot be excluded as it would be necessary for supporting publish/subscribe services [23] where users subscribe for a given content that will be asynchronously produced in the future. Many of these services are implemented, for example, by means of HTTP long polling strategies, which make extensive use of long (potentially unanswered) requests. [49] presents a complete discussion on the best practices about timer setting and on how 'long' these requests should be. In general, 30 s is considered a safe value, as longer timers may be undesirable for intermediate proxies placed between the server and the client. However, solutions with longer values are widely adopted in common web applications (e.g.,FaceBook and some web-based mail applications, which to our experience often use timers of more than one minute). In addition to that, we have to consider that one or more malicious users could craft artificial requests with the purpose of

filling the available PIT memory on routers, thus implementing a Distributed Denial of Service (DDoS) attack.
With PIT overflow, users would see their Interests discarded by routers and, consequently, they would experience an increasing rate of retransmissions until the network completely collapses. With this problem in mind, some possible PIT architectures have been proposed with the aim of reducing the required table size. A first simple solution [37] is to realize the PIT by means of a hash-table, where each URI is encoded using a fixed number of bits. Other possible solutions deploy more complex architectures. For example, [16] proposes a tree-like structure to arrange the PIT entries in order to also ensure high lookup, insertion, and deletion speed, while [93] presents a very efficient PIT architecture based on Bloom Filters.
While these solutions may be effective during normal operation, performance degradation might be experienced when the system is under the abovementioned DDoS attacks. This kind of attack may be implemented by distributedly generating Interest packets that contain a valid destination prefix but non-existing resource names, so that routers properly forward Interests and store new entries in their PITs, but responses never come back. The destination might possibly be colluded with the attacker and simply drop incoming packets. In order to maximize the impact of the attack, a malicious user could request always different resources thus avoiding Interest merging. Furthermore, the attacker could also select large values for the LifeTime field. Figure 1.6 shows a very simple network consisting of three routers,


PIT R2

URI	Arrival Interface
/it/badDest/xjos....fhufsh/	eth1
/it/badDest/gjnv....hfhfbv/	eth1
/it/badDest/vcee....fbdfd/	eth1
...	...
/it/badDest/qpkf....fopwe/	eth1

PIT R1

URI	Arrival Interface
/it/badDest/xjos....fhufsh/	eth0
/it/badDest/gjnv....hfhfbv/	eth0
/it/badDest/vcee....fbdfd/	eth0
...	...
/it/badDest/qpkf....fopwe/	eth0

Figure 1.6: Attack scenario

an attacker (or, equivalently, a bot net), a normal client and a bad destination, which is placed ad hoc by the attacker. The role of the bot net is that of generating Interests towards the fake destination. Routers along the dashed path (R1 and R2) are flooded with unanswered Interests, which indeed increases their memory usage. This work presents an analysis of the PIT resilience to possible overload and consequent service disruption. A comprehensive simulation study of the possible PIT architectures is done in order to evaluate their performance from this perspective.

1.4 An overview on the existing literature

The feasibility of the CCN approach with regard to the amount of requested resources at nodes, together with a proper CCN node dimensioning, has been recently object of a wide research activity. For example, [3] presents an efficient router design and describes some possible usage scenarios. Designing an efficient forwarding plane is also the subject of [94], which identifies key issues related to the protocol fast speed implementation and establishes some principles to be observed in order to design scalable forwarding architectures. Furthermore, [63] presents a feasibility study of CCN and concludes that CCN nodes based on current technologies would still be unable to sustain requests arrival rates at the Internet scale. However, they can sustain rates at Content Distribution Network and small ISP scale. Specifically concerning the PIT dimensioning problem, two recent papers propose different architectures conceived for reducing the PIT size. As also described in the previous section, [16] proposes a tree-like PIT structure, while [93] present a PIT architecture based on Bloom Filters. However, both papers do not provide a quantitative evaluation of the PIT overload problem and do not consider possible DDoS attacks. Concerning the security threats in a CCN network, the available literature mainly covers privacy, data authentication, and data integrity issues. For example, [73] addresses the privacy problem deriving from the stateful operation of CCN routers, while data authentication and integrity have been considered since the seminal CCN papers [37, 38]. Instead, a comprehensive study of the possible PIT overload issue deriving from a DDoS attack is missing. [17] presents a solution to mitigate these attacks, while the ongoing work by Gasti et al. [65] gives an introductive overview of the problem and of some ongoing experiments.

1.5 PIT Resilience Analysis

This section focuses on the evaluation of the PIT resilience to possible overload. Our analysis is performed by simulation and considers three possible PIT architectures: (*i*) a PIT storing all the bytes that compose an URI, referred to as SimplePIT; (*ii*) a PIT storing fixed length entries evaluated as hash values of the URIs, referred to as HashedPIT; (*iii*) a PIT implemented through multiple Bloom Filters placed in each router interface, as described in [93]. This last solutions is known as DiPIT. The PIT architecture presented in [16] is not explicitly investigated here as it shares with the HashedPIT the same principle of introducing fixed length entries (in [16] the numerical codes that make up the logical tree structure are reduced to fixed length).

While in the first two cases PIT overloading can be measured in terms of memory occupancy, with possible memory overflow, in the latter case this is not possible as the DiPIT is based on Counting Bloom Filters, where memory occupancy is constant. In essence, the filter fills the entire memory size and each Interest is encoded in a sequence of '1s' to be added at given positions of the filter. In this case, PIT overloading results in a high rate of the so called false positive events, namely, the node erroneously concludes that an Interest is present in the filter and either does not propagate it or erroneously removes some entries from the PIT when a chunk is received. In both cases this results in a degradation of the service perceived, as both events require one or more Interest retransmissions. [93] proposes an effective combination of filters to reduce the false positive probability during normal operation (see [93] for further details), but this may increase when the network is under attack. In order to quantitatively evaluate the effect of false positives and also enable a coherent comparison among the three considered PIT architectures, we base our analysis on the average percentage of Interest retransmissions that occur at users' machines. This is clearly proportional to the average download time they experience, and hence it is a metric of the overall network performance.

A further issue to address is the selection of a proper simulator complying with the requirements of our study. Some CCN simulators are available, but all are customized for specific analyses. For example, ccnSim [22] has been designed mainly for the evaluation of cache replacement techniques and hence is not optimized for experimenting on the PIT management. Hence, in order to focus our analysis on PIT issues and meet our specific requirements, we developed a full custom event-driven Java simulator.

1.5.1 Simulation scenario

With the main aim of obtaining significant and quantitative results, the network configuration and user behavior models are refined to faithfully recreate realistic scenarios. In particular, we adopt as a reference the network of Telecom Italia, a prominent Italian ISP. Data related to the Telecom Italia network that are of interest in this context are publicly available on the web.

First of all, the topology adopted in our simulations reproduces the real structure of the Telecom Italia network [78] (see Figure 1.7, represented at Point of Presence (PoP) granularity). The network is divided in two areas: the first one that covers Northern Italy and the second one that covers central and Southern Italy. PoPs are almost equally divided between the Northern and the Southern areas and in our simulator each PoP is represented by a router. This topology is typical also of many

ISPs in Europe.
Concerning the network population, we adopt the number of broadband Internet subscriptions currently active in the Telecom Italia network - around 9 million - available in the ISP investor relation [80]. We also assume the topological distribution of users among the ISP PoPs to be directly proportional to the geographical distribution of the ISP customers. As a result, we allocate users to various POPs according to Figure 1.8, that contains the fraction of users per Province derived from real statistics. This allows us to accurately reproduce the download traffic pattern of the ISP users. Furthermore, access bandwidths are considered uniform for simplicity and equal to 7 Mbps (download) and 1 Mbps (upload), which are the values characterizing a large percentage of Telecom Italia users. The overall header size of the protocols underlying CCN is assumed fixed to 20 bytes.

Regarding the selection of the content to be retrieved by clients, we assume a Zipf-Mandelbrot probability distribution, which Saleh et al. [72] proved to properly model the behavior of users in a content distribution P2P network. Given the content delivery nature of a CCN network, this probability distribution could reasonably model the user behavior also in our case.

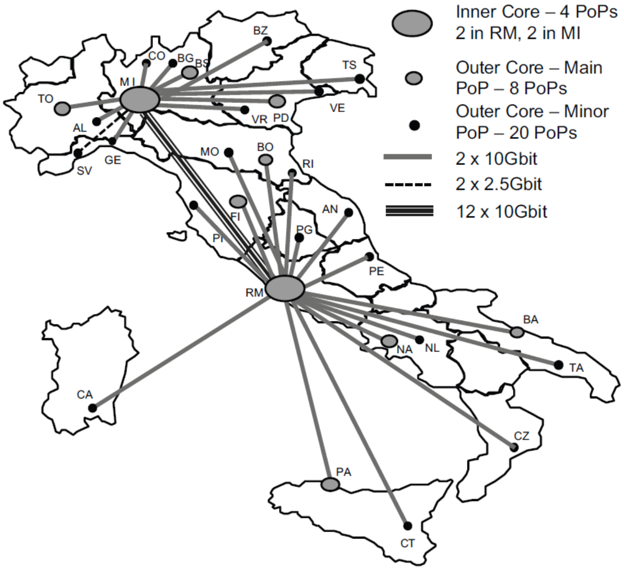


Figure 1.7: Telecom Italia topology

According to the Zipf's Law and given the total number of resources in the network, which are uniformly distributed among per-PoP content providers, the corresponding distribution function is expressed as follows:

$$p(i) = \frac{1}{(i+q)^{\alpha}} \qquad \forall i \in [1, N]$$

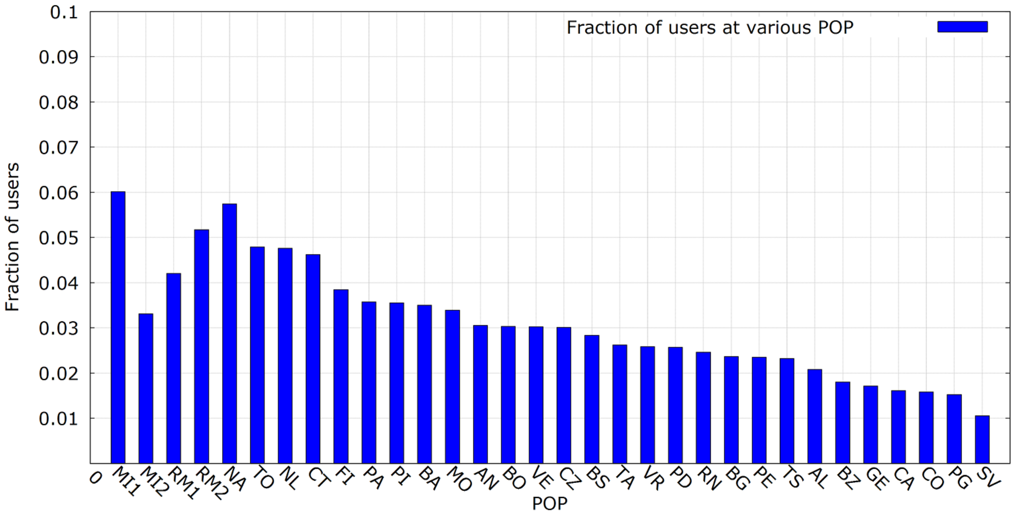


Figure 1.8: Users allocation

where $p(i)$ is the probability of extracting the i-th content available in the network, q and α are two parameters that [72] fixed to $\alpha = 0.55$, $q = 25$ for a residential ISP (as Telecom Italia is), and N is the total amount of resources. The resulting function is depicted in Figure 1.9 where we highlight in green the theoretical trend of the mathematical expression above mentioned and, in red, we highlight the density function obtained by our custom random generator assuming 1.000.000 samples generation. The code implemented to realize the Zipf distribution is shown in Listing

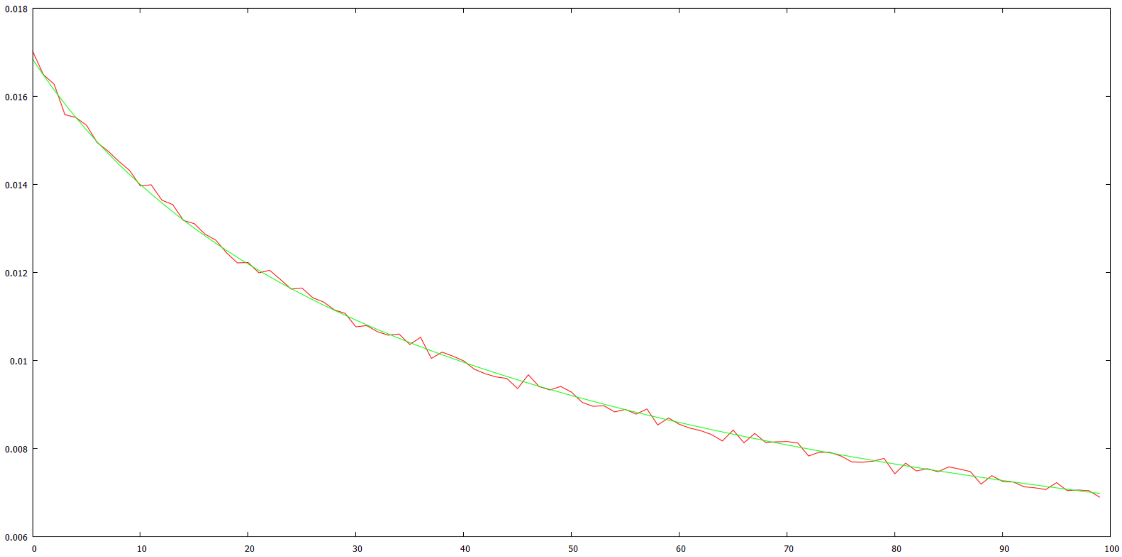


Figure 1.9: Zipf probability distribution (density function)

1.1.

Listing 1.1: Zipf random generator

```
public class ZipfDistribution {

        private Double[] cdf;
        private Double[] density;

        private Random random;
        private int totalElements;
        private double alpha;
        private double q;

        public ZipfDistribution(long seed, int totalElements, double alpha, double
            q)
        {
                this.totalElements = totalElements;
                this.alpha = alpha;
                this.q = q;
                random = new Random(seed);
                init();
        }

        private void init()
        {
                Assert.assertTrue(q > 0);
                cdf = new Double[totalElements];
                density = new Double[totalElements];
                double temp, norm=0;
                int i;
                for(i=0; i<totalElements; i++)
                {
                        temp=(1.0/Math.pow(i+q, alpha));
                        Assert.assertTrue(temp > 0);
                        cdf[i]=temp;
                        density[i]=temp;
                        if(i!=0)
                                cdf[i]+=cdf[i-1];
                        norm+=temp;
                }
                for(i=0; i<totalElements; i++)
                {
                        cdf[i] /= norm;
                        density[i] /= norm;
                }
        }

        public int nextInt()
        {
                double r = random.nextDouble();
                int i;
                for(i=0; i<totalElements && r>=cdf[i]; i++);
                return i;
        }

        public double getProbability(int i)
        {
                if(i < 0 || i > totalElements)
                        return 0.0;
                return density[i];
        }
        public Double[] getDensity() { return density; }

        public Double[] getCumulativeFunction() { return cdf; }

}
```

Download requests are modeled using a Poisson process with average rate equal to 500 requests per second. This results in an average value of around 12 million simultaneously active downloads in the steady state. We believe this network load is even higher than that observed in a 9 million users network during normal operation and hence it is significant for our study.
Another key parameter for our evaluation is the memory availability at CCN nodes. Reasonable values for the PIT size that match both the required memory access speed and current memory technologies are some hundreds of MB [93], available by means of the SRAM technology. In order to be more conservative and also with the aim of analyzing a possible near future scenario, we fix the PIT size to 1 GB. In the DiPIT case, this value refers to the overall available filters memory (the PITs plus the shared Bloom filter). In order to optimize the DiPIT implementation and make the simulation more scalable, we implemented a model of the Bloom Filter rather than the *actual* Bloom Filter (see Listing 1.2).

Listing 1.2: DiPIT implementation details

```
package it.polito.ccn.simulator;

import it.polito.ccn.simulator.exception.FullPITException;
import java.util.List;
import org.apache.commons.math3.distribution.UniformRealDistribution;

/*
 * Bloom Filter model implementation
 */
public class DiPIT extends PIT{

        public static final int HASHES = 4;      // number of hash functions

        private static UniformRealDistribution dist;

        private PIT p;
        private int falsePositives;
        private long totalQueries;
        private final long size;

        static{
                dist = new UniformRealDistribution();
        }

        public DiPIT(double pitSize) {
                super(pitSize);
                p = new RegularPIT(1000*pitSize);                        // Big
                    enough
                size = (long) (pitSize * Math.pow(10, 6));        // Used for prob
                    calc
                falsePositives = 0;
                totalQueries = 0;
        }
        @Override
        public int getAverageEntrySize() {
                return p.getAverageEntrySize();
        }

        @Override
        public boolean existsUri(Uri uri, int fragmentNumber) {
                totalQueries++;
                if(p.existsUri(uri, fragmentNumber))
                        return true;
                double prob = getFalsePositiveProbability();
                double random = dist.sample();
                /* False positive */
                if(random <= prob)
                {
                        falsePositives++;
                        return true;
                }
                else
                        return false;
        }

        @Override
        public List<Interface> getInterfacesPending(Uri uri, int fragmentNumber) {
                return p.getInterfacesPending(uri, fragmentNumber);
        }

        @Override
        public boolean addEntry(Uri uri, int fragmentNumber, Interface iFace)
                        throws FullPITException {
                return p.addEntry(uri, fragmentNumber, iFace);
        }

        @Override
        public void removeEntry(Uri uri, int fragmentNumber) {
                p.removeEntry(uri, fragmentNumber);
        }

        private double getFalsePositiveProbability() {
        return Math.pow((1 - Math.exp(-HASHES * (double) p.getElementsInPIT() /
            size)), HASHES);
    }

        public void clearAll() { p.reset(); }

        public int getElementsInPIT() { return p.getElementsInPIT(); }

        public int getFalsePositives() { return falsePositives; }

        public long getTotalQueries() { return totalQueries; }

}
```

In particular, we simulated the false positive probability by means of the following analytical model:

$$p = (1 - e^{-\frac{K*n}{size}})^K \tag{1.1}$$

where:

- K is number of hashing functions used. We set K=4 in our simulator;
- n is number of elements added to the filter;
- size is the total size of the filter (1GB in our case).

Thanks to this implementation we are able to speed-up the computation since there is no implementation of the different hashing functions and also to save a massive amount of space. Considering the network of TI, we would have 30 DiPIT (one for each router) leading to 30GB of ram only for these filters (not considering the edge of the network). In this case, the simulation would have required too many resources while the mathematical model is able to scale well also on common laboratory machines.

Concerning the attack parameters, we consider a maximum aggregate attack bandwidth of 4 Gb/s, which is a realistic value for a medium scale ISP such as Telecom Italia [68], and Interest LifeTimes values that vary between 4 s (the default value considered in the CCNx [62], the prototypal implementation of CCN node) and 180 s. Notice that those are reasonable LifeTime values if we think to enable complete publish/subscribe services in a CCN network, as discussed in Section 1.3.

1.5.2 Results

This section reports on the simulation results obtained for the three considered PIT architectures: the SimplePIT, the HashedPIT and the DiPIT. We consider a distributed bot net sending Interests to fake destinations connected to the Rome edge router. Hence, our analysis focuses on the Rome PoP as it is the most overloaded node. In essence, we keep track of the memory usage at the Rome PoP. Furthermore, we also measure the average percentage of retransmissions experienced by users, as depicted above.

I scenario - SimplePIT

In the first scenario, we consider to deploy a SimplePIT at each router. This PIT implementation stores the entire URI in the memory so it is the simplest architecture we can think of. In order to maximize transmission efficiency, and consequently maximize attack impact, it is important for the attackers to craft very long URIs. We selected 1000 bytes in our simulation. In particular, each malicious URI has a valid 13 bytes prefix (e.g. **/it/badPubRm/** required for reaching the destination) and a non-existing resource name whose length is set to a randomly selected string of 1000 bytes.
Simulation results concerning the memory occupancy at the Rome PoP and the related retransmission rate are shown in Table 1.1 for several values of the overall bot net bandwidth and of the fake Interest LifeTimes (retransmissions are referred

Attack settings	**SimplePIT**		**HashedPIT**		**DiPIT**	
	Retransm.	RAM Usage	Retransm.	RAM Usage	Retransm.	RAM Usage
Band = 100 Mbps *LifeTime = 4 sec*	0	≈ 49 MB	0	≈ 25 MB %	≈ 0.01 %	1 GB
Band = 500 Mbps *LifeTime = 4 sec*	0	≈ 245 MB	0	≈ 125 MB	≈ 2.42 %	1 GB
Band = 2 Gbps *LifeTime = 4 sec*	0	≈ 980 MB	0	≈ 500 MB	≈ 87.6 %	1 GB
Band = 4 Gbps *LifeTime = 4 sec*	≈ 15 %	≈ FULL	≈ 83 %	≈ FULL	≈ 90 %	1 GB
Band = 100 Mbps *LifeTime = 60 sec*	0	≈ 735 MB	0	≈ 375 MB	≈ 21 %	1 GB
Band = 100 Mbps *LifeTime = 120 sec*	≈ 37 %	≈ FULL	0	≈ 750 MB	≈ 86 %	1 GB
Band = 100 Mbps *LifeTime = 180 sec*	≈ 52 %	≈ FULL	∞	≈ FULL	≈ 88 %	1 GB

Table 1.1: PITs performance evaluation

only to finished downloads). Users do not experience a considerable retransmission rate until the PIT is completely full. However, by increasing either the overall attack bandwidth or the Interest LifeTime, routers start to be unable to correctly handle incoming traffic and all the connections are significantly slowed down.
These results can be analytically justified. Let us consider the following case as an example (we recall that the overall headers of underlying protocols is fixed to 20 bytes in our simulations):

$$B_{attackers} = 2\,Gbps, \quad LifeTime = 4sec \tag{1.2}$$

$$|Interest|_{attackers} = 1033\,bytes(\overbrace{1013}^{\text{URI}} + \overbrace{20}^{\text{HEADER}}) \tag{1.3}$$

The number of Interests per second generated by the bot net can be calculated in the following manner:

$$\frac{(2 * 10^9)\ bps}{(1033 * 8)\ bits} \equiv 242013\,\frac{Interest}{sec} \tag{1.4}$$

Considering a 4 s LifeTime, we obtain:

$$\approx 242013 * 4 = 968.052\ entries \Rightarrow \approx 980MB\ occupied \tag{1.5}$$

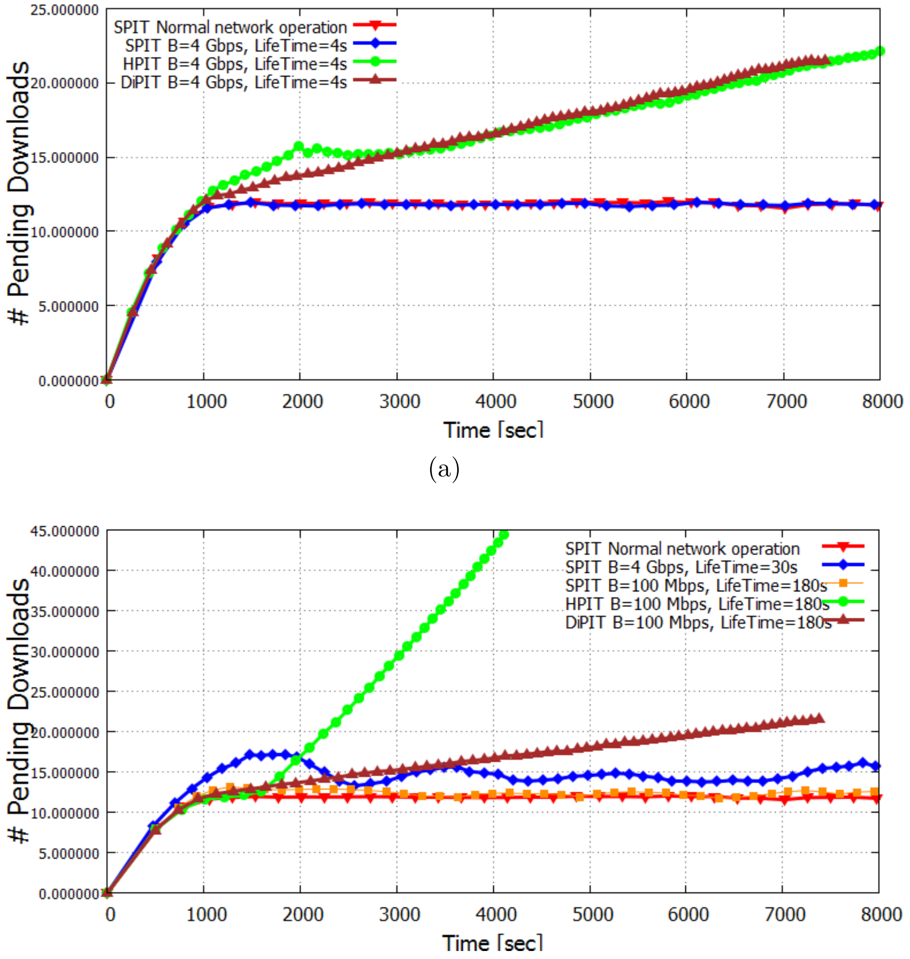
which is consistent with our simulation result.

II scenario - HashedPIT

In the second scenario, we consider the HashedPIT, namely, a centralized hash table storing a fixed length entry for each URI in transit. We exploit the SHA-1 hashing algorithm in order to ensure a negligible collision rate. In this context, the best size for an attacker's URI is 20 bytes, according to the SHA-1 output digest (160 bits). Longer URIs are useless as would be reduced to 20 byte strings by CCN nodes. In this case the resultant attackers transmission efficiency, here intended as the amount of transmitted data that will be stored in the PIT with respect to the total traffic generated by the host, is around 50% due to the non-negligible underlying header size (20 byte headers in our simulations lead to 20 / (20+20) = 50 % transmission efficiency). Since in the previous case the efficiency was about 98% (1013 byte long URIs lead to 1013 / (1013 + 20) ≈ 98%), one could infer that an attacker has simply to double its bandwidth to get the same impact of the SimplePIT case. This is only partially confirmed by simulation results presented in Table 1.1. We can observe how, as expected, the memory occupancy is halved at equal simulation conditions and, consequently, the retransmission rate grows slowly with the increase of the attack intensity. However, it is worth noticing the greater attack effectiveness when the PIT is almost completely full with respect to the previous case. Such a situation here prevents users from finishing any downloads and the number of retransmissions tends to infinite, namely, no useful data are received by clients, who continue to retransmit Interests: the network is completely unusable. The infinite value in Table 1.1 just represents the situation in which no stable value is reached for the number of retransmitted packet from the client point of view. This is due to the fact that the first data structure is more complex to overfill as stored Interests are of highly variable length and attackers do not have exact information of the amount of memory that is still available at routers. Hence, when the PIT is almost full many attackers' fake Interests are discarded as they are too large with respect to the available space. Some shorter users' Interest can instead be accommodated and hence the network is still able to operate, although with a significantly reduced efficiency. In the HashedPIT case, instead, 20 bytes is always the best choice for URI length so the attack is more effective and destructive. This clearly does not mean that the SimplePIT can be considered resilient to DDoS attacks as specific attacker techniques may be adopted to possibly saturate the PIT. This analysis is left for future work.

III scenario - DiPIT

The last scenario refers to a very different PIT and router architecture. The central PIT is split into multiple smaller per-interface PITs, each implemented by a Counting Bloom Filter data structure. The specific description of this proposal and the algorithm adopted is available in [93]. The retransmissions observed in this case are due to the Bloom Filter false positive events, as described above in this section. The



(a)

(b)

Figure 1.10: Network performance evaluation

probability for a Bloom Filter to return a false positive may be approximately evaluated[2] as $(1-e^{-\frac{k*n}{m}})^k$, where k is the number of hash functions deployed to code a given element, n is the number of elements currently in the filter, and m is the total size of the filter. In our simulator each Bloom Filter is modeled by means of this probability function. Furthermore, the specific architecture based on different filters and the insertion/deletion algorithms presented in [93] are reproduced in order to obtain consistent results.

In [93], the authors also suggest possible values for k, which vary according to the Interest arrival rate that the router has to support. We set this value to 4 hash

[2] Assuming independence for the probabilities of each bit being set.

functions because larger values are not suitable for high-end routers. For the Bloom Filter, we assume for simplicity 8 bit counters and no counter overflow. Simulation results are presented in Table 1.1. It is worth noticing how although the DiPIT does not suffer from memory overflow (neglecting counter overflow), false positive events become a truly limiting factor when the network is under attack and many entries are inserted into the filter. In fact, considerable retransmission rates are observed also with non-huge attack intensities.

1.5.3 Discussion

These simulation results lead us to a twofold conclusion. First, none of the analyzed PIT architectures is overloaded during normal operation in the considered network scenario. Even with a low intensity attack, memory usage is reasonable and no retransmissions are observed. This in some way confirms that even a traditional SimplePIT-based solution might be currently deployed at ISP scale, as also concluded by the analysis in [63]. However, second, there are significant weaknesses in all the architectures when the attack intensity grows.

To summarize our results, Figure 1.10 plots the number of pending downloads in the network over time for some reference values of attack bandwidth B and fake Interest LifeTime. In Figure 1.10(a), the considered attack intensity leads this number to slowly diverge with a similar trend for both the HashedPIT and DiPIT based nodes. For the SimplePIT we do not observe a significant gap with respect to the system behavior during normal operation. Figure 1.10(b) considers a more critical scenario: B=100 Mbps, LifeTime=180 s and also B=4 Gbps, Lifetime=30 s for the SimplePIT. We can observe how the SimplePIT is only partially affected in this second case, while the system fast exits the steady state due to retransmission rate divergence when the HashedPIT is deployed. We can conclude that the HashedPIT is the architecture most affected by the considered attack, while the SimplePIT is the architecture most resilient for the reasons explained above. The DiPIT has an intermediate behavior. Clearly, these results hold in our attack scenario. Starting from this point, one could design other specific attacker behaviors that even worsen the perceived service, especially for the SimplePIT. For example, an attacker may:

- (i) combine broad bandwidth and higher LifeTime to increase attack effectiveness;
- (ii) distribute more zombies around the network to avoid attack source detection;
- (iii) exploit more bad prefixes in order to make any countermeasures even more complex to deploy.

This is a non-exhaustive list that further motivates the real need for proper countermeasures to DDoS in a CCN network. Our results show how all architectures are affected by these attacks and hence further studies are needed to figure out possible countermeasures. One can be, for example, the introduction of smarter algorithms for LifeTime management at content routers, which adapt LifeTimes as a function of the network load: a router can grant larger LifeTime values in case of low traffic congestion and, conversely, implement a sort of LifeTime shaping when the load increases. With such a mechanism, we would break down the hypothesis that intermediate nodes do not manage LifeTime values as well as PIT entries removal (for example by discarding the ones which have a too long expiration time). Other mechanisms could be based on an Interest RED (Random Early Discard) strategy based on the amount of occupied memory. The higher the congestion, the larger is the probability of discarding an incoming Interest.

1.6 Interest Flooding Attack (IFA) Countermeasures and Solutions Assessment

In traditional IP networks, DDoS attacks usually plague end terminals since the connection information state is kept by these devices. On the other hand, CCN is hardly based on the fact that intermediate routers maintain per packet state. This feature allows the protocol to avoid routing loops since each Interest is recorded into the PIT table, and also to implement native multicast support because each node remembers who asked for what. However this feature arms attacking users because, as we will show in this section, there exists the possibility to artificially generate forged packets with the only aim of wasting router memory. In particular, let us consider the scenario depicted in Figure 1.11

The attacker needs to announce a valid prefix to send fake Interests to; in our example, /com/badContent/* serves to this purpose and let us call that node *prefix Hijacker*. In addition, the attacker needs one or more zombie clients (or even a large

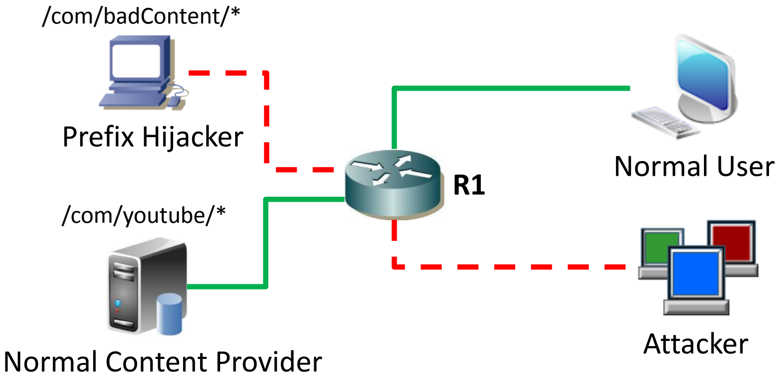


Figure 1.11: Interest Flooding Attack example

botnet) to start sending Interests targeted at the existing prefix but with non existing (and possibly long) resource names, for example /com/badContent/abcdefg…z. Such packets will correctly reach the prefix Hijacker, i.e. the machine configured to receive these datagram without generating any response, but no useful data will be sent back. With this simple procedure, all the Interests seen by R1 (and for which R1 creates an entry in the PIT) will remain in the device memory until the timeout, called LifeTime , expires. As better analyzed in the next section, an attacker with a proper bandwidth may be able to deny the service for legitimate users whose Interests would correctly call back Data packets but are not able to be accommodated due to memory exhaustion. We can roughly compute the overall memory usage due to the attacker activity with the following formula:

$$Mem_{bytes} = \frac{Band_{bps}}{8} * Lifetime_{sec} * Trasm_{eff} \tag{1.6}$$

$$Trasm_{eff} = \frac{f(namelength)}{Totalpacketlength} \tag{1.7}$$

where:

- Band is the attacker bandwidth;
- Lifetime is the time interval set in the fake Interest;
- $Trasm_{eff}$ is the transmission efficiency intended as the amount of data within the packet that goes in the PIT (over the total transmitted data).

It is worth mentioning that the efficiency can have a significant variation, depending on different factors. For example, it depends on the amount of transmitted data that ends up in the PIT, usually proportional to the name length, and it depends on how the router stores those data in the table, if it applies any kind of hashing function or uses special encoding scheme. We represent any possible scenario in (2) with a generic function f to model the possibility to apply any processing to the name string. To simplify our analysis we assume that the router stores each forwarded request as a plain text in the PIT and we neglect other implementation dependent details such as additional per PIT entry state information. We will also assume that fake Interest packets contain very long names in order to increase the above mentioned transmission efficiency and consequently maximize the attack impact.

The IFA is analyzed in several papers, which also discuss some possible solutions. For example, [88, 28, 14, 90] contain a preliminary evaluation of possible security threats applicable to the CCN/NDN architecture. Notice that the DDoS we are dealing with in our work is just one attack among different possible variants. In fact, some attacks are targeted at wasting the router CPU resources while others aim at poisoning content through caches. The possibility to implement these attacks in a real network has been widely recognized as a concrete scenario, thus encouraging the research area focusing on solutions and countermeasures. To the best of our knowledge, three main proposals are targeting the specific problem of IFA. The work by Afanasyev et al.[1] presents a framework named Satisfaction Based Pushback to limit the number of forwarded Interests for a given prefix depending on statistics about the Interest satisfaction ratio, i.e. a metric representing the number of Interests that bring back useful Data in a given time interval, as opposed to those Interests that expire without causing the delivery of useful data. Another countermeasure, referred to as Poseidon, is presented in [15]. This approach is similar to the previous one in that it gathers statistics about the traffic seen at each router but with a different activation procedure, as we will show in the following. The last solution we will consider, referred to as Traceback, is described in [17]. The idea is to activate a countermeasure after the memory usage has reached a predefined threshold. The algorithm consists in generating spoofed Data packets for the entries that are causing the memory overflow problem. Collaborative messages between the involved routers can be exploited to obtain better results in terms of reactivity in all these designs. Our goal is to evaluate and compare them in our use case scenario, i.e. the network of the main Italian service provider.

1.6.1 Countemeasures to DDoS attacks

Satisfaction based pushback

The Satisfaction based pushback algorithm is described in [1] and it works as follow: each router computes the Interest satisfaction metric as the ratio between the number of satisfied Interests over the number of forwarded Interests. This computation is performed on a per interface basis and it is an indication about the probability of an Interest coming from a certain interface to be satisfied. Such metric can be directly used to calculate the limits of Interests the router is willing to forward from each interface. After the computation, the router announces its limits to downstream neighbors in order to rate limit the incoming traffic, especially for those interfaces that are increasing the burden on the PIT memory occupancy. After some time, the router computes new statistics and clears its history with an exponential decay, in order to restore the original limits and give a chance to each interface to have more virtuous Interests forwarded again. Notice that metrics about the traffic may be collected at different granularities: prefixes, FIB entries and so on. The traffic limit messages announced by each router will flow back up to the end user interface. Whether or not the client node obeys to the imposed limit, the rate limiting will anyway be enforced by each router on the path. Notice that this approach is made available by two features of the CCN protocol: (i) the state information maintained by each router can be exploited to gather some statistics about the traffic and (ii) the symmetric data routing implied by CCN can be used to compute which Interests call back data since they will follow the reverse path.

Interest Traceback

The Traceback algorithm is described in [17] and it is designed to release the unwanted PIT entries when the available amount of memory space falls under a predefined threshold. The detection phase is rather simple and only requires to monitor the PIT memory usage over time. After detecting an abnormal memory occupancy, the Traceback process is triggered and a set of spoofed Data packet are generated for those entries that remained unsatisfied for a long time. The spoofed Data packets carry the name needed to satisfy the offending Interests and are forwarded downstream to release resources all along the path. In order to leverage, as much as possible, the memory space available to the PIT, we defined the threshold after which the Traceback is activated as 90% of the occupied memory. Such aggressive limit avoids algorithm overreacting and allows the network to support temporary traffic peak without triggering any Interests blocking mechanism. Since some implementation details were omitted in the reference paper, we designed our code to meet as close as possible the countermeasure description. In particular, in Algorithm 1, we show the code of the monitoring process which is scheduled every second to check if the memory occupancy is over its alarming value.

Listing 1.3: Traceback monitoring process

```
void Traceback::CheckMemory()
{
    IF pit_size greater than MAX_PIT_size*90/100
        // Look for fake entries and send spoofed Data
        Traceback::FindAndSend();
    END IF
    SCHEDULE next check in 1 sec
}
```

If it is the case, the FindAndSend() function is invoked to generate spoofed Data packets and make them travel towards the attack initiator. A simplified high level vision of our implementation can be seen in Algorithm 2.

Listing 1.4: Traceback sending spoofed Data

```
void Traceback::FindAndSend()
{
    FOR EACH Entry in Pit
        IF IsOld(Entry)
            FOR EACH Face in Entry.FacesList
                IF Face.IsConnectedToEndUser()
                    BLOCK Face
                ELSE
                    GENERATE SpoofedData
                    SEND SpoofedData through Face
                END IF
            END LOOP
            RELEASE memory
        END IF
    END LOOP
}
```

Poseidon

Poseidon[15] is a framework to mitigate the effect of the IFA on CCN/NDN networks. It shares some similarities with the previous approach since it also collects statistics by observing the forwarded traffic. The main difference is in the detection phase: Poseidon is triggered when two metrics exceed their corresponding thresholds. The two parameters used by Poseidon are defined as follows:

$$\begin{aligned} \omega(r_i^j, t_k) &= \frac{\#\text{Interests from } r_i^j \text{ at time } t_k}{\#\text{Data from } r_i^j \text{ at time } t_k} && (1.8) \\ \rho(r_i^j, t_k) &= \#\text{ of PIT bytes used by } r_i^j \text{ at time } t_k && (1.9) \end{aligned}$$

In order for Poseidon to be activated, both of these two metrics must exceed the allowed value. The algorithm uses both in order to limit the number of false positive, namely the number of times it erroneously detects an in progress attack. To this end, it is very important to accurately tune the temporal window over which these metrics are calculated. In fact, a small window value would result in raising an alert too soon even when the threshold have been exceed just for a short physiological network burst. On the other hand, a too large value would cause the algorithm to react too slowly. In our simulations we use a one second window, in order to obtain an acceptable tradeoff. Poseidon reacts to an attack detection by imposing limits on the number of accepted Interests from the interface which exceeded both thresholds and lowering them for that interface. Additionally, collaborative messages are exchanged by the routers on the attack path to share information about their state. After some time, if the traffic becomes normal again, Poseidon will restore all the thresholds to their original values and the imposed limits are deactivated.

1.6.2 Simulation Scenario

Our simulation scenario is the network of the main Italian service provider, Telecom Italia (TI), whose logical topology is publicly available[78] and is showed in Figure 1.7, with PoP granularity. The connection between each user and its corresponding POP is modeled as an ADSL line with 7Mbps/1Mbps downlink/uplink bandwidth because these are very common values for TI domestic DSL contracts. The total number of customers is around 10 million and their distribution in the network is coherent with the population density of each province. Another important parameter is the PIT size. In this sense, the technology used to implement this table can deeply vary, depending on the traffic rate to be supported. For example, supporting the traffic of a national backbone router may require the use of static RAM technologies to fit the requirements in terms of access time, while other peripherals devices may employ larger (but slower) DRAM to serve less intensive traffic patterns. Since our focus is on high end network appliances, we choose a default value of 1GB for the maximum PIT size, which represents a sort of upper limit with respect to current hardware technologies. See [63] for a deeper insight on this topic. To load our network, we implement download arrivals at each client side and limit customers to download just one file at a time, for simplicity and scalability of the simulations. Each file to be requested is selected among the global resources catalog with a Zipf probability distribution having α=0.55 and q=25 as in [86]. This traffic load represents our baseline for all the simulations. The attacker model is rather simple since it generates Interest packets at the maximum speed allowed by its uplink bandwidth. Each packet contains a different (and random) resource name to avoid Interest merging on routers along the path. We distributed many attackers around the network, targeting the same prefix in order to concentrate the effects on a central device, which, in our scenario, is the Rome PoP. For this reason and also for the sake of brevity, we provide results and metrics only for this network appliance. One prefix Hijacker node is directly connected to the Rome PoP to attract all the fake Interests and discard them. Such behavior makes PIT entries unsatisfied for the whole Lifetime thus wasting precious memory portions.

1.6.3 Simulation Results

In order to have a baseline for our simulations, i.e., the behavior of the network when no countermeasure is implemented, we run a simulation campaign varying the attack bandwidth starting from a minimum value of 100 Mbps up to 4 Gbps. Notice that this attack bandwidth is perfectly feasible since many security reports[68] confirm the possibility to obtain an aggregate attack bandwidth even higher than 10 Gbps by exploiting distributed zombies, hence our assumption is quite conservative. The results of this preliminary test are showed in Table 1.2;

Attack Bandwitdh	**Retransmissions**	**RAM usage**	**# Downloads**
0 bps	0 %	0.2 %	2841000
100 Mbps	0 %	5 %	2841000
500 Mbps	0 %	25 %	2841000
2 Gbps	0 %	98 %	2841000
4 Gbps	5.14 %	99.9 %	2101500

Table 1.2: Baseline network performance with no countermeasure deployed

Simulations results are provided either in terms of memory performance (RAM usage) and also in terms of the overall network functioning (percentage of retransmissions and total number of completed downloads). We report the RAM usage as the amount of memory occupied just by the PIT in a stable situation, i.e., after any transient has disappeared. As it can be seen, in all the cases in which the PIT is not completely filled the number of downloads completed by normal users is constant. Thus, the network can be considered 'stable', namely, all the transfers are not significantly affected by the attacker activity. On the other hand, in the last case (last row), the clients and their connections are considerably slowed down and the overall number of finished transfers decreased. This is in line with our expectations since the fake traffic has two consequences: (i) wasting part of the links bandwidth and (ii) fulfilling the PIT, which becomes unable to admit regular Interests. This implies an increasing number of Interest retransmissions, computed as:

$$retr_{\%} \quad = \quad \frac{\#Interest - \#Data}{\#Interest} * 100 \qquad (1.10)$$

where:

- #Interests is the actual number of Interests sent;
- #Data is the number of Data composing the file in transfer.

In an ideal scenario, the number of transmitted Interest packets per file transfer should be equal to the number of Data packets, leading to 0% retransmissions. This computation is performed for each finished download and then averaged at the end of the simulation. After implementing the countermeasures in the network, we run a simulation campaign for each of them and obtained the results depicted in Table 1.3. We start our analysis with the Pushback algorithm. As we can see from the results, an increasing attack bandwidth causes a worse network performance, especially considering the overall number of downloaded files. The surprising result is that the countermeasure limits the network also in case of low intensity attacks because the algorithm is designed to compute the maximum number of acceptable Interests and announce it to downstream routers. Since the fake Interest packets mix with normal requests, the resulting limit computed for the interfaces of the routers along the path and targeted by the attacker involves also part of the legitimate traffic. The worse performance cannot be captured in the retransmissions computation since, as previously mentioned, it is performed only for finished file transfers thus not taking into account downloads in progress at the end of our simulations. For what concerns the Traceback framework, results are definitely better and almost all the files are correctly delivered. Only in the last case, with an aggregate attack bandwidth of 4 Gbps, some downloads are not completed. The reason is that we have some transients between each attack detection and the countermeasure deployment, so that some regular Interests are initially discarded by on path routers. After reaching the threshold set for the Traceback process (this only happens in the last two

Attack Band	Retransmissions			RAM Usage			Total downloads		
	PB	TB	POS	PB	TB	POS	PB	TB	POS
0 bps	0 %	0 %	0 %	0.2 %	0.2 %	0.2 %	2841k	2841k	2841k
100 Mbps	0 %	0 %	0 %	5 %	5 %	0.3 %	2839,5k	2841k	2841k
500 Mbps	2 %	0 %	0 %	[0.7-6.4]%	25 %	0.3 %	2707k	2841k	2841k
2 Gbps	0 %	0 %	0 %	[0.7-6.4]%	0.2 %	0.3 %	2706,5k	2841k	2841k
4 Gbps	0 %	13 %	0 %	[0.7-6.4]%	0.2 %	0.3 %	2706,5k	2837,5k	2841k

Table 1.3: Countermeasures simulations results with different attack bandwidths

rows where the attacker band is equal or higher than 2Gbps), the countermeasure is triggered and the spoofed Data immediately release the memory wasted on intermediate nodes. The routers, which give connectivity to end users as well as attackers, quickly identify the attack originator and lock the link. In this way the attacker activity is completely denied and the network operating restored, as proved by the good performance achieved in terms of finished downloads and memory usage. The number of retransmissions is slightly higher because, during the transient, some downloads may experience a slow down. The last framework under test is Poseidon. As evident from Table 1.3, simulation results are even better than the previous cases as confirmed by the total number of finished downloads, which is completely restored in all the considered attack scenarios. This is a positive consequence of the dynamic behavior of Poseidon and its combined usage of two parameters, ω e ρ. To improve the performance, we set the maximum threshold for the metric related to the memory destined to each face, by taking into account the link bandwidth to which that interface is connected, as follows:

$$p_i^j = \frac{B_j}{\sum\limits_k B_k} * MAX\_PIT\_SIZE \tag{1.11}$$

where:

- p_i^j is the maximum amount of memory for the PIT entries coming from interface j on router i;
- B_j is the bandwidth of the link connected to interface j;
- $\sum\limits_k B_k$ is the sum of all the bandwidths of links connected to the router interfaces.

In this way, bigger links (as in the case of the RM-MI link) are allowed to use a larger amount of the RAM portion because they usually serve a wider part of the traffic hence it is reasonable to have more PIT entries due to those interfaces.

The considered thresholds are automatically lowered while the attack is starting, resulting in less probability for the attacker to have its Interests forwarded upstream. After the statistics become normal again (with an exponential decay law), thresholds are raised again to progressively reopen the link to the attacker. However, this oscillation is never dangerous for the network performance as the system performs an early detection of this phenomenon thanks to the monitoring process that is triggered with 1 second frequency, and the regular traffic is definitely not affected by the fake traffic. This last result reveals that Poseidon is the most resilient framework against IFA and can successfully shield the considered network topology under the assumptions made for the attacker behavior.

1.7 Scalability: issues and challenges

Since its first appearance, CCN has gained an ever increasing interest thanks to the revolutionary idea of addressing data rather than hosts, in a world where information (e.g. web pages, media content and so on) is becoming more and more important. In all ICN proposals the notion of location-dependent host identifier has been completely abandoned in favor of a name based routing scheme: users interested in a given content can simply generate a request indicating the resource name, without caring of its exact location. This vision has nurtured a number of research work, ranging from the study of smart cache management algorithms[25, 12, 89] to the exploitation of CCN for mobility environments[44] and high availability applications. However, the inherent pull nature of the protocol poses serious challenges for all the applications that work in a push-like fashion since CCN establishes a precise direction for all network communications: indeed, the requesting node is also the connection initiator and the content provider has apparently no direct way to start sending data. Nevertheless, this behavior is exactly the one that many modern web applications realize since, in many cases, they may need to notify the clients with an asynchronous event (like the reception of an incoming message or an alert triggered by some external module). In this case, the data transfer initiator is the node holding the data (e.g. the web server) and not the one who is interested in it (e.g. the client browser). In traditional IP networks the standard Http Get/Response (which works in a much similar way as the CCN pull model) has been enhanced with additional protocol extensions to support server to client data transmission (Web sockets, HTTP streaming, long polling techniques and so on). We argue that allowing those functions also in CCN would greatly augment the appeal of the data centric architecture.

What we have talked about so far, i.e. the automatic update of the client side when the content producer generates news for it, shares many similarities with what, in the literature, is referenced to as a publish/subscribe system[23], namely a network architecture allowing multiple clients (i.e. the subscribers) to subscribe to multiple topics, whose data is generated by other entities called publishers. In principle, when a publisher makes a content available to the world, all the interested subscribers should be notified for the newly generated content and the fresh data is pushed to them. This idea is much general and can be applied both to traditional TCP/IP networks and also to ICN. Indeed regarding the latter, there exist some proposals that aim to implement the pub/sub architecture within the CCN infrastructure and we will discuss them in Section 1.7.2. However, while these solutions work very well when the number of content generators is small and not so rapidly changing, there is still little clarity on other scenarios where the content generation is extremely rapid and data providers/consumers appear and disappear very frequently in different parts of the network, especially for what concerns scalability at the Internet scale. For instance, let us think to an alarm signal broadcasting a real time message to all the interested users or a webmail service informing its clients for a new incoming message. Enabling the pub/sub system to work well in such dynamic scenarios, would enable push-based applications to be implemented on top of CCN as well.

The main contribution of this section is to explore some possible designs for a generic push application, underlining the main challenges that must be solved and proposing an effective way to cope with them. While doing this, we will use a specific case study application in order to make our study more concrete. In particular, we consider the design of a CCN based social network application since it allows us to consider the problem of rapidly connecting/disconnecting clients, such as mobile clients, and also to evaluate the mechanisms that can be leveraged to deliver asynchronous notifications to clients. This context is significant as already discussed in [46, 92, 91, 60]. We propose a design based on the idea of location-dependent host identifiers which make each host involved in any form of dynamic push application, able to receive automatic update from a content source. This idea is derived from well known concepts inherited from the TCP/IP world. However, to the best of our knowledge, their applicability to CCN has never been considered while, in fact, it can bring new benefits and advantages to the CCN protocol. In our work, we evaluate and compare our solution with COPSS[13], the state of the art framework enabling the instantiation of a publish/subscribe environment on CCN. Our simulation results are based on the topology of the most prominent Italian service provider (i.e. Telecom Italia). We collected a number of statistics about the Facebook social network since it is widely recognized[47] as the most famous social website in the world, and we use them to represent a realistic network population in our experiments.

1.7.1 Problem statement and requirements

Given the functioning of the CCN pull mode, we may infer the following points: (*i*) the connection initiator is the one who wants to receive a particular content, (*ii*) the entity which has the content (e.g. a content provider) cannot send any data without being asked first. However, since our focus is on applications that need to notify clients on an asynchronous basis, the connection initiator is, in principle, the one holding the content. For example, considering a modern web application implementing a web mail service, it would be necessary for the server to notify the clients for each new incoming message and this notification must be delivered in real-time as not to deteriorate the quality of experience perceived by the end users, as already discussed in [4, 48]. This behavior is achieved in standard HTTP (over IP) by means of different techniques. One of the most popular and easy to implement is the one realized by AJAX requests. With this technique, the client polls the server for a new update by means of a background AJAX connection. The server leaves the channel open until it has some news for the client, then sends them and terminates the connection. Then, the client side reissues another request in order to always make a channel available to the server for further data pushing. This mechanism is summarized by the sequence diagram in Figure 1.12. It is worth mentioning that other programming techniques may achieve the same result of the just described one. The set of these techniques is generically called COMET programming and it includes HTTP streaming, AJAX requests, long polling and so on. Since the introduction of HTML 5, there is also a special interface for bidirectional communications which is called WebSocket API. As we have stated above, including this programming mode, which is very common in the current scenario of network applications, within the boundaries of the CCN infrastructure seems to be as important as complex. In the next sections, we discuss possible designs and their relative challenges with a special attention on the scalability of the transport network.

1.7.2 An overview on existing solutions

In CCN we have some proposals that target push applications. For example, [16, 83, 86] mention the idea of a long Interest lifetime useful to keep an open channel between the content provider and the data consumer. With this approach, the content provider can simply generate the response (when it is ready) by exploiting the pending Interest; Dai et al.[16] envision the use of a special type of never dying Interest which is meant to remain in the PIT even when the Data has been forwarded. The idea is much similar to the above one, except for the fact that clients do not have to re-express the Interest when it expires. COPSS[13] relies on a complete and well designed pub/sub environment, where a Rendezvous Node is in charge of centralizing the subscriptions to the interested topics (from the clients) and the publications of new pieces of data (from the publishers). This idea shares some similarities with the IP multicast implementation. We will give a larger importance to COPSS in the following sections since, to the best of our knowledge, it

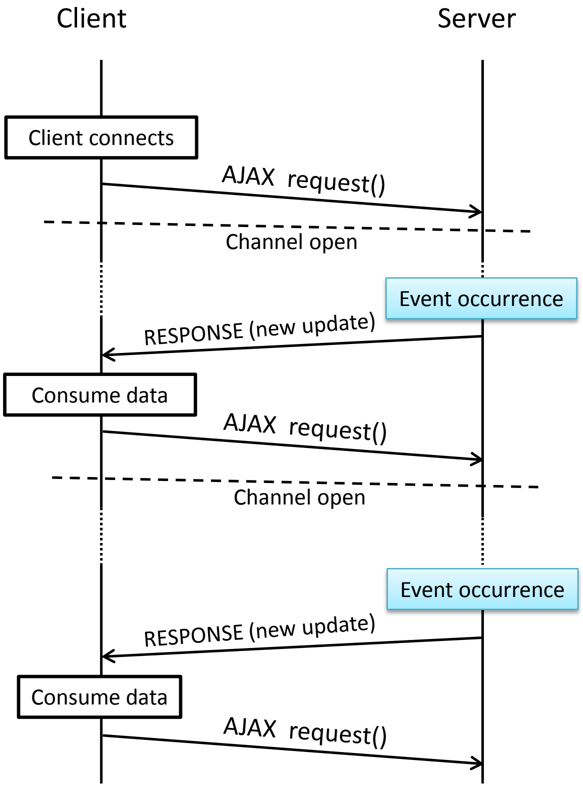


Figure 1.12: AJAX with long polling

is the state of the art technique for pub/sub within the CCN proposal and it includes a variety of solutions to the common problems of such network architectures (push enabled content dissemination, efficiency, possible incremental deployment, offline subscribers management, and so on). A possible alternative approach, which appeared in the CCNx project mailing list and also in the NDN website FAQ[59], requires the server to send out an Interest to the client meaning that it wants to receive an Interest for a newly generated content. This scheme must be carefully evaluated since it introduces additional packets which represent a sort of overhead to signal the availability of a fresh content.

The first two solutions (Interests with long timeout and never dying PIT entries) are theoretically possible but they are not completely scalable since they require to store a massive amount of state on intermediate routers for a possibly long time. The situation can even get worse in case of malicious attacks, as many recent papers demonstrated[90, 86, 1, 15, 88, 17, 20]. For this reason, we do not consider them in our evaluation section, as we are interested in Internet scale applications (like a social network application is).

There is some additional related work in the area. In particular, [8] provides a significant contribution from a design point of view since it proposes an effective way to implement push/pull communications by means of a unique, unified data structure: the Forwarding Information Base (FIB). The architecture envisioned is general and not specific to CCN while our target is exactly CCN. Moreover, [74] highlights and discusses the importance of enabling real-time content distribution over Named Data Networking but an extensive evaluation of possible mechanisms to accomplish this in a realistic network topology and workload conditions is missing. Finally, [54] is centered around the idea of deploying a social network application over the ICN infrastructure. To support this idea, the authors underline the big advantages that can be obtained by exploiting the idea of in-network caching. However it does not fully evaluate the impact on a very large scale in terms of state to be maintained on network routers.

In addition, it is worth mentioning that there are some important European project [19, 26] aiming at designing a clean slate approach to the pub/sub implementation exploiting ICN concepts and ideas. While these proposals are much general and refer to ICN, in this work we will focus on a particular incarnation, that is CCN.

1.7.3 Push Architecture Designs

We now introduce and describe two possible architectures to implement push applications on CCN. In particular, we adapt COPSS[13] which is targeted at realizing a network of content publishers and subscribers, since it is the state of the art framework for the pub/sub paradigm over CCN, and we also implement our solution based on a special Interest sent from the content publisher to a specific consumer by means of an ad hoc host identifier. This last design, fully compatible with the widely recognized CCN design[37], has been derived by various discussions which appeared on some of the main CCN-related mailing lists with additional key mechanisms we have designed to better support dynamic nodes (e.g. clients) and increase the scalability of the routing architecture.

Copss

COPSS[13] is a well known framework implementing a publish/subscribe system over CCN. The COPSS architecture is based on the concept of Rendezvous Node (RN). In essence, the RN is a “special" type of COPSS-aware router which is in charge of centralizing the reception of subscriptions from all the subscribers interested in a given range of contents (identified by one or more Content Descriptors (CD)), and the reception of news from content generators spread all over the network. In order to balance the load among different network devices, the network may be configured to have different RNs which handle different CDs. A valid CD could be, for instance, “*/sport/soccer/italy*".

All the network routers must have a FIB entry in order to properly forward the subscriptions (carried in a packet called Subscribe[3]) to the relative RN and they must store this information in a data structure called Subscription Table (ST) which encapsulates all the subscriptions and the interfaces from which they arrived.

The content generation is handled by a special type of transmission unit called Publish packet which is conceptually similar to a standard CCN Data packet (it is meant to transport the content payload) but it is routed through a FIB matching procedure instead of being forwarded by means of PIT breadcrumbs. When reaching the destination RN, that is the Rendezvous node designated to handle the target CD, the Publish packet is propagated to downstream subscribers, thanks to the STs scattered in the network, so that they can consume the new update.

From this brief description, we can deduce that the RN becomes the routing tree root for the CDs it is configured to handle. In this sense, all the packets (both Subscribe packets and Publish packets) must go first to the RN and only after that, they will be delivered to the leaves. As previously mentioned, many RNs can exist to manage different descriptors. This can lead to sub-optimizated paths since all the Publish/Subscribe packets for a given CD must be firstly routed to the corresponding RN, as showed in Figure 1.13. Therefore a trade-off between load distribution and path optimization is necessary. In our evaluation section we assess this aspect with particular emphasis on our topology (the Telecom Italia network).

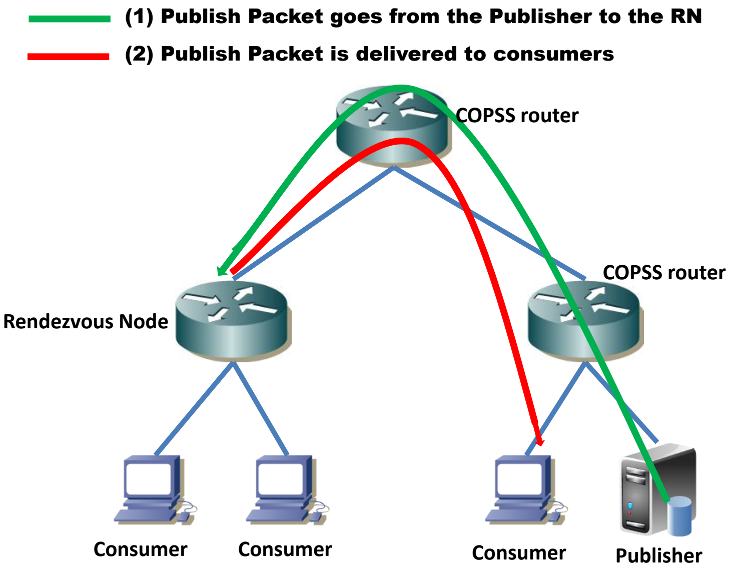

(1) Publish Packet goes from the Publisher to the RN
(2) Publish Packet is delivered to consumers
COPSS router
COPSS router
Rendezvous Node
Consumer
Consumer
Consumer
Publisher

Figure 1.13: A tree-like COPSS network

[3] Conceptually, the Subscribe packet is very similar to an Interest packet except for the fact that it includes a set of contents.

Location-Dependent Host Identifiers (LDHI)

In this section we present our proposal. We start by introducing the basic pattern upon which it is based then we move forward and introduce the additional enhancements that make the architecture more feasible and scalable. The starting pattern is the following one: each time the server has an update for a particular client, it sends out an Interest to a specific prefix the user is able to attract Interests for, that is a prefix agreed by the user and the network and for which the user has registered a FIB entry. This packet serves just to communicate the availability of a new content. Part of the name contained in the packet could also be exploited to carry the information about the newly generated content. At the reception of this packet, the client, in turn, sends an Interest to the announced name in order to immediately retrieve the fresh content.

To clarify the operations of this model, let us consider the example of Figure 1.14: Alice is connected to a generic social network application and she wishes to receive all the updates from her friends. Bob is one of them. When the server has a new content (due to Bob social activity), the server sends an Interest with the following name: “*/mysocialapp/Alice/mysocialapp/Bob/news/123456*". The first part, namely “*/mysocialapp/Alice/*" is a globally routable prefix which Alice is able to attract Interests for. The second part, namely “*/mysocialapp/Bob/news/123456*" is the name of the update so that Alice is enabled to easily strip it and explicitly request the new content by means of a classic Interest/Data transaction. This basic scheme is theoretically possible but it introduces some drawbacks that we must closely evaluate. First of all, it needs two additional packets for the setup stage. This introduces an extra RTT compared to the IP solution, where the data

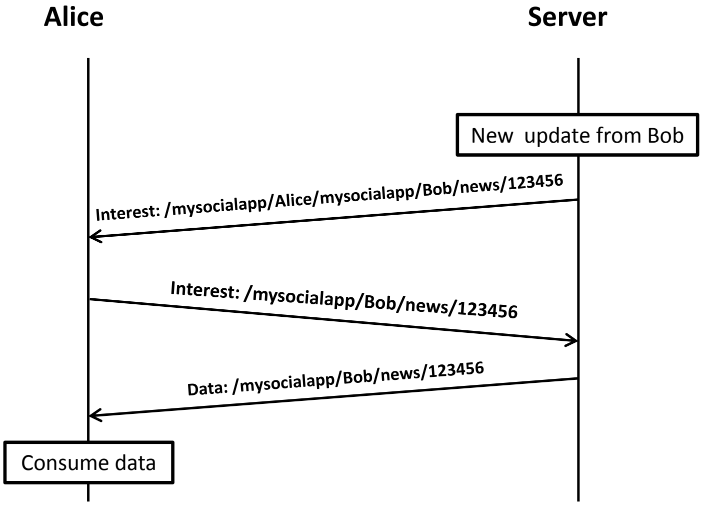

Alice
Server
New update from Bob
Interest: /mysocialapp/Alice/mysocialapp/Bob/news/123456
Interest: /mysocialapp/Bob/news/123456
Data: /mysocialapp/Bob/news/123456
Consume data

Figure 1.14: Basic pattern used to push data to Alice

is immediately returned in a half RTT. In addition to this, we have another concern about the reachability of clients. In fact, the above mentioned scheme is based on the fact that the prefix "*/mysocialapp/Alice/*" is "globally routable". This is theoretically feasible but practically un-scalable. In fact, the semantics of this prefix is location independent in the sense that the client must be reachable to this prefix from any portion of the network it is connected to. This implies a continuous update of the routers' FIB entries as soon as a client changes its position and also a small probability of aggregating entries.

Having in mind the scalability (especially of the transport network), we re-designed this basic scheme and propose to provide client nodes with an identifier which should be *location-dependent* and closely related to the place/network where the user connects. In our vision this identifier should be provided by the network itself by means of an ad-hoc protocol in a much similar way to the Dynamic Host Configuration Protocol (DHCP) protocol, which provides IP addresses in current LANs. Those identifiers can be leveraged to increase the scalability of the FIB because they aggregate on a geographical basis like IP addresses in traditional networks. Our idea is conceptually different from the ones available in the literature. For example, in the Voice-over-CCN implementation[38], clients are reachable through domain specific names, which solely depend on the service provider name space. We propose to change this vision according to the geographical location of the user so as to exploit aggregability of FIB entries. Given this idea of location-dependent host identifiers (LDHI), the previous scenario with Alice connected to the social network application, changes as in Figure 1.15.

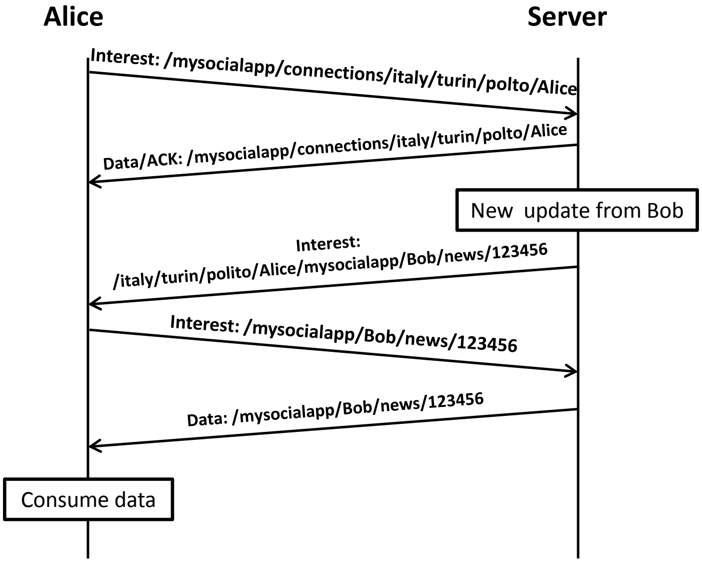

Alice
Server
Interest: /mysocialapp/connections/italy/turin/polto/Alice
Data/ACK: /mysocialapp/connections/italy/turin/polto/Alice
New update from Bob
Interest:
/italy/turin/polito/Alice/mysocialapp/Bob/news/123456
Interest: /mysocialapp/Bob/news/123456
Data: /mysocialapp/Bob/news/123456
Consume data

Figure 1.15: Example of how to push data to a client based on location dependent host identifiers

As it can be noticed from the diagram, each user needs a packet to make the server aware of its current identifier. This can be achieved with a single Interest packet at service connection time. In our example the connection Interest carries the name "*/mysocialapp/connections/italy/turin/polito/Alice*": the first part, namely "*/mysocialapp/connections/*", is a globally routable prefix which identifies the service endpoint taking care of user connections. The second part, namely "*/italy/turin/polito/Alice*", is the dynamic information about the prefix that can be used to send Interests towards the client and we call it location dependent host identifier. In this way, the server can easily extract this prefix and store it locally. At this point, it should be clear that, whenever Alice connects from different parts of the network, she will get different node identifiers, depending on the geographical position of the hosting network, and she will have to notify the server about its current unique identifier. After the connection, the server application is able to contact Alice by sending an Interest addressed to "*/italy/turin/polito/Alice/mysocialapp/Bob/news/123456*", which includes the name of the new content. When receiving this Interest packet, Alice knows a new update has been generated by Bob and it can be obtained by a traditional Interest/Data transaction targeted at "*/mysocialapp/Bob/news/123456*".

With our proposal, we achieve the following benefits: the network does not have to update the FIB entries of all the routers in the network[4] each time a new client connects and the FIB becomes extremely scalable thanks to this IP-inherited idea of geographical aggregation of entries. Clearly, we must evaluate the overhead introduced by the initial setup stage needed for each notification delivery. This is the main purpose of the evaluation section, through which we quantify the additional traffic impact.

It is worth noticing that many concepts and ideas explained in this section are explicitly inherited from the IP world, especially the notion of host identifiers (see [30] for a full discussion on the use of node identifiers in mobility environments). However, their applicability to CCN has never been discussed because this new network architecture is exactly aimed at *abandoning* this approach based on a host-centric vision of the network. This intuition is well motivated and justified by current trends in Internet growing since users are currently more interested in retrieving specific contents rather than reaching specific network nodes. Nevertheless, we would like to underline that this concept of location-dependent identifiers does not change the way CCN works and its core idea of addressing data rather than physical locations. We see this mechanism as an *additional* tool to implement applications that need to reach a particular host, thus realizing a conversational communication. The rest of the content dissemination must be performed with the standard Interest/Data procedure which should not involve any location-dependent identifier. We look at

[4] Or a portion of them, depending on how much the position changes and on the routing protocol.

this proposal as a significant extension to enlarge the CCN applicability also to the specific services we are considering in this section.

1.7.4 Evaluation

Introduction

In this section we tackle the complex problem of evaluating the impact of the above described solutions, either in terms of network traffic and latency and also in terms of memory requirements for intermediate nodes. The complexity is due to at least two reasons: first of all, we have to consider a realistic network topology in order to understand the implementation issues in a concrete scenario. Second, we must conjecture a plausible network load, meaning the number of users connected to the network, their bandwidth, the number of notifications per second received by them and a set of other statistics representing the users behavior. In the next two subsections, we will discuss exactly these aspects.

The metrics adopted to compare the solutions pertain to the number of packets (both Interest and Data packets), the latency introduced by the two approaches, and the memory usage on intermediate nodes. These elements represent the key dimensions of our problem. Since in COPSS, the memory requirement can only be expressed (in this context) in terms of the ST size, we compare this data structure with the PIT in the LDHI scenario. We believe this is a fair comparison since these are the two most overloaded protocol components in the experiments we performed.

The topology

Our simulations are fully focused on the network of Telecom Italia (TI, the main Italian provider) since we believe the structure of this provider is quite common to most middle-sized network operators. The logical network topology of TI is publicly available and it is showed in Figure 1.7, at PoP granularity. The PoPs presence is almost equally distributed between the northern part of the country and the southern one, serving the most densely populated provinces. Rome and Milan are the core cities for the entire topology and they are connected through an high-speed backbone link.

We have distributed TI users in all the provinces (according to the demographical size of the province itself) by means of one edge router per PoP. The link between each user and the edge router is 7 Mbps for the downlink and 1 Mbps for the uplink as these are the most common values for Italian domestic ADSL. Other bandwidths are set as in Figure 1.7.

Users model

Our focus is on a generic social network application implemented on top of CCN. For this reason, we have gathered statistics[77, 76] about the users behavior, targeting the major social network website, that is Facebook. We then deduced all the necessary parameters in order to achieve accurate simulation results. We summarize them in Table 1.4.

Table 1.4: Facebook Statistics

#	**Statistic**	**Value**	**Description**
1	TOTAL_FB_USERS	1.3 bilion	Total number of Facebook users in the world
2	TOTAL_IT_USERS	26 milions	Total number of Facebook users in Italy
3	CONTENT_UPDATES	939.000	Number of content updates every 60 seconds
4	AVG_TIME_SOCIAL	2H 29M	Avg time users spend on social media per day
5	AVG_VISIT_DURATION	20M	Avg duration of a single user visit
6	AVG_FRIENDS	130	Avg number of friends per user

From stats 1 and 2, we deduce that the number of Italian user is 2% over the total FB population hence we can use this information to infer the number of content updates per second generated by the Italian population (derived from stat 3). From stats 4 and 5, we can compute the average number of visits per day per each user and from stat 6 we can finally generate a complete social graph by extracting AVG_FRIENDS friends per each FB user.

Each time a content is generated, we randomly select the corresponding author from our population and the fresh update is delivered to all of its friends. This reflects the behavior of most social networks, which consists in updating the client web page with all the friends news feed.

Social networking over COPSS

The first part of our evaluation consists in simulating the network illustrated in the previous sections, with an adapted implementation of COPSS. We have implemented this architecture in ndnSIM[2], the reference NS-3 module for NDN/CCN simulations.

We have assumed the server node is connected to the Rome PoP and it acts as the node who receives all the updates generated by the various users and it publishes them through the COPSS network. Notice that this choice optimizes the paths because the RN and the application server are adjacent.

All the clients act as subscribers thus each user must subscribe to the topics it is interested in. Since we are implementing a social network application, we assume the topics correspond to the user profiles a client wants to receive news from. This procedure, i.e. the subscription to all friends updates, must be performed each time a client appears in the network so as to inform the pub/sub network of its presence. A symmetric procedure has to be applied at user disconnection time since the network (in particular the Subscription Tables) must be updated to correctly handle the user departure. To solve this specific problem, we have assumed an explicit Unsubscribe packet which is sent out from all the clients when they disappear from the network. This is a simplification since, in a real deployment, COPSS aware routers should handle clients disappearing without any announcement. This can be accomplished by using a soft state-like mechanism which is able to recognize the client departure, for example by means of a timeout event. For the sake of simplicity, we have assumed that all clients behave correctly, i.e. they always communicate their disconnection.

Since each client has 130 friends (on average, as in stat 6) we expect to have 130 Subscription packets on average at each node entering the network, each one representing the willing to receive friends updates. The same applies at disconnection time for the unsubscription procedure. Notice that, due to the assumption that Unsubscribe packets are always generated, we are simulating an extra traffic. However, if a timeout mechanism existed to avoid this assumption, a massive amount of packets would have been generated to periodically refresh ST entries in COPSS aware routers, hence the network traffic is balanced in these two possible implementations.

Social networking through location dependent identifiers (LDHI)

The second part of our evaluation is focused on the architecture including location-dependent host identifiers (LDHI). Regarding this point, a client connected for example to the Rome PoP would have an identifier similar to "*/rome/users/roClient123/*". The relative PoP has a single entry, that is "*/rome/users/**", pointing to the interface the edge router (ER) is connected to. Therefore, assuming each client is connected to a different interface on the ER, this last has, in turn, as many entries as the number of users since the aggregability is not obtained at the ER level but only in the transport network.

The service connection process is performed at each client connection thanks to the client sending an Interest packet, meant to announce its current prefix to the server. Such packet contains a name like "*/facebook/connections/rome/users/roClient123/*". The first two components are parts of a globally routable prefix and the rest ("*/rome/users/roClient123/*") can be stored by the server application for future communication with that particular host. Notice that this connection process is much lighter than the COPSS case because we only need to send a single packet to join the social service while in COPSS the client must send a Subscription Packet for each friend profile he is interested in. This reasoning applies also for the disconnection phase since a single packet addressed to a similar name (i.e. "*/facebook/disconnections/rome/users/roClient123/*") is enough to leave the social application in LDHI.

We assume that the server portion of the social network application fully knows the social graph, which is the list of friends for each connecting client. This is a reasonable assumption to reflect the functioning of modern client-server applications that tends to concentrate all the important information on the server side.

We now introduce the steps used to deliver asynchronous notifications to the end users. Let us assume that a certain client (for example, the one identified by "*/venice/users/veClient2413*") generates a new content and transfers it to the server. At this point, the server has the task to push data towards all the interested clients. This information (the list of all interested clients) is explicitly contained in the server state as the social graph holds all the users interconnections. Given the friends list, the server takes the corresponding identifier of each user and generates an Interest packet. For example, assuming that the user possessing "*/milano/users/miClient2413*" as node identifier is a friend of the user having "*/venice/users/veClient2413*" as node identifier, the server generates an Interest with the following name: "*/milano/users/miClient2413/facebook/news/%00%01%CB*". The first three components are necessary to reach the correct destination (the user willing to receive the update) while the rest is the NDN name of the newly generated content. In our simulations, we simply name all the content updates with a sequential number incrementally increased by the server (the very last name component). At the reception of this Interest, the client side of the social application understands that a new interesting content is available and requests it with a classical Interest/Data transaction targeted to the content "*/facebook/news/%00%01%CB*". With the proposed scheme, we also benefit from the CCN caching capability since all the notified client nodes ask for the same content (in a short period of time) hence the content diffusion is greatly aided by the CCN protocol caching features.

Simulation results

Simulation results are depicted in Figure 1.16, where we show the amount of packets generated by the COPSS and LDHI solutions. In both cases, we only report the traffic seen at the Rome PoP since it is the most overloaded device and also because it serves as the Rendezvous Node in COPSS. With this choice, we have the RN and the FB server very close to each other, hence paths are perfectly optimized. The first relevant difference is in the number of outgoing Interests (or Publish packets in COPSS), that is the amount of Interest/Subscribe packets sent from all node interfaces. In fact, as we can see from Figure 1.16(b), COPSS does not propagate any packet while LDHI constantly generates traffic for the whole simulation time. This is not surprising since the role of the RN in COPSS is to concentrate all the subscriptions without further forwarding them. In LDHI the outgoing Interests are packets coming from users asking to connect/disconnect to/from the service and packets asking for updates. In addition, we also have the initial Interests exploited by the server to setup the communication for each notification transfer.

Another key point is related to the number of incoming Interest/Subscribe packets, which is higher in COPSS. This is due to the massive amount of packets that are sent each time a user enters/leaves the social network and it is proportional to the average number of friends per user (in both procedures).

Regarding the amount of Data/Publish packets generated, we do not have any appreciable difference since network caches eliminate any possible delay between one client content request and the other ones in LDHI. This is a beneficial effect of having a network of caches deployed throughout the topology.

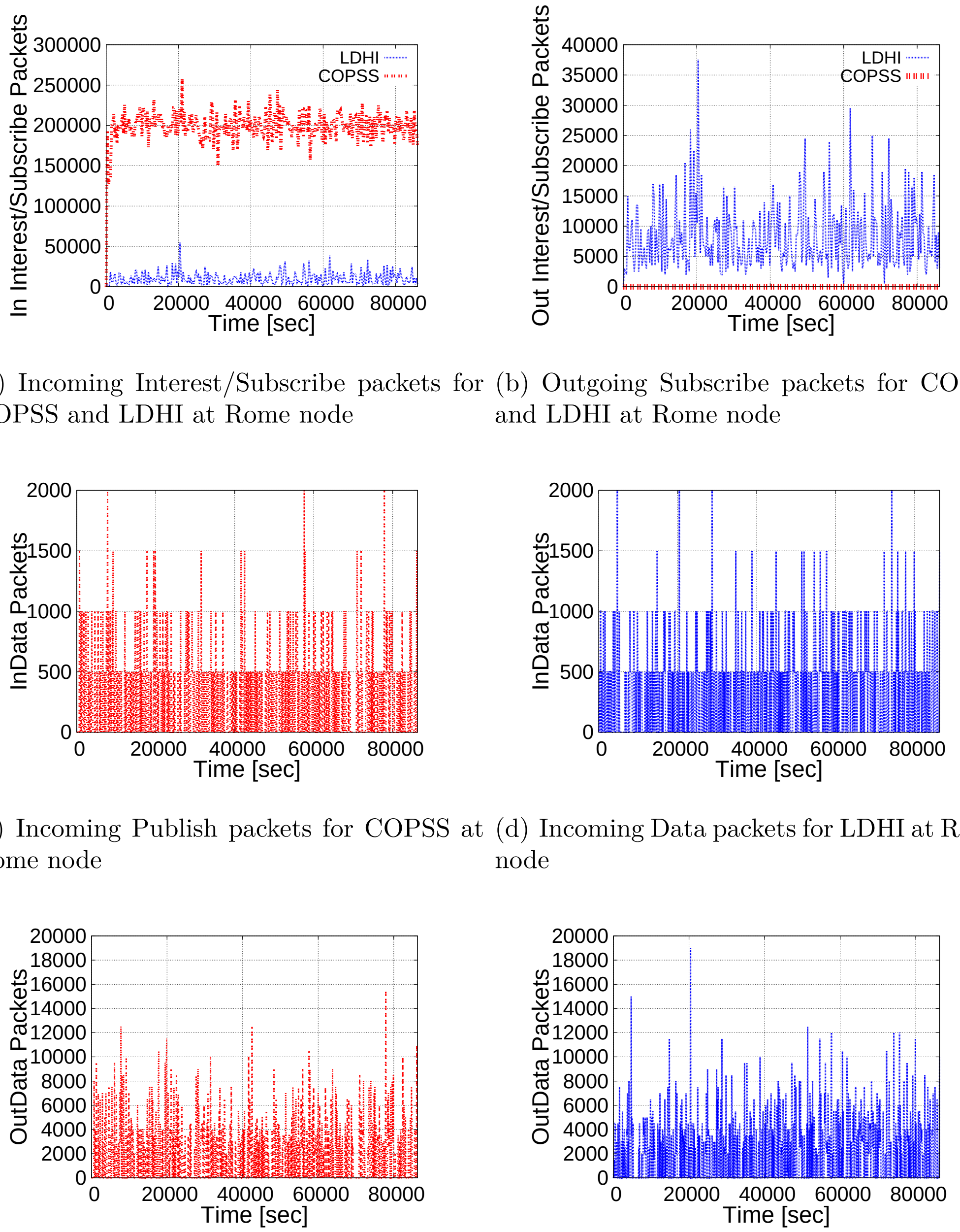
Evaluating the performance of the two systems in terms of latency experienced by the end users when a new content is delivered to them, is also very important. Here the latency is defined as the time between the production of a content update by the server and the reception of the news by the client side. As we underlined in the previous sections, LDHI has an higher latency time due to the setup stage needed for each status update. This phase takes an extra RTT and has an impact on the overall amount of time needed by the client to receive a content. In Figure 1.17, we highlights the performance results obtained in our simulations and we show the probability distribution for different latency values. From the comparison, it is evident that COPSS outperforms LDHI, requiring three times less time to deliver a fresh content on average. One of the main reasons to accept this drawback is that many applications do not have so strict real time data delivery constraints (social networking falls under this category). Furthermore, we proved LDHI has the positive effect to generate (under the assumptions made) less traffic and this can be an essential feature in the mobility scenarios, where users are limited in the amount of traffic they can generate.

As our simulation results show, the initial concern about the additional overhead introduced in LDHI by the setup stage, is partially mitigated thanks to some of the implementation details we have described so far. Clearly, the extra RTT needed to transfer data still exists with LDHI (and not with COPSS) but, depending upon the kind of application we want to implement, this latency can be neglected. Our understanding is that many applications (web-based or not), might have no need for a low latency data delivery hence delays of some milliseconds may be tolerated (like in the social network example). For what concerns the global traffic produced in the network, we can conclude that LDHI performs better in this network scenario (the overall number of generated packets is smaller). Of course, this conclusion holds in our context. One may find other contexts where the population behavior is less penalizing in COPSS, depending on the number of content subscriptions per second, connections/disconnections per second, and so on. However, we can argue that our result is significant as the topology and the application behavior are both derived from the real world.

The traffic produced and the latency introduced by the two solutions are just the first dimensions of our problem. Another important and crucial aspect we would like to analyze pertains with the amount of internal state that must be maintained by intermediate routers. In this sense, we provide in Figure 1.18(a) and 1.18(b) the RAM usage we experienced with the two solutions on the Rome PoP, since it is the central (and most overloaded) switching device of our topology.

A possible solution could be the distribution in the network of several RNs to balance the load in terms of state which has to be maintained. However, we consider this option inappropriate in this specific context for the following reason: since we are implementing a single application with a single domain prefix (i.e. "*/mysocialapp*"), the balancing could be done on a specific CD basis. For example, CDs with "*/mysocialapp/users/a**", may be handled by one RN, while other CDs, e.g. "*/mysocialapp/users/b**", may be handled by another RN, and so on. The problem of this approach is that we cannot know in advance which would be the optimal placement for RNs location since we do not know a priori where clients interested in CDs matching "*/mysocialapp/users/a**" come from. Furthermore, the clients location is likely to change in a very rapid manner, thus an optimal RNs configuration would probably exist only for a small time interval. In fact, we recall that all Publish/Subscribe packets must go first to the relative RN. Hence, leaving the path construction to the randomness of users connection may be not a wise idea. These considerations lead us to the conclusion that RNs load balancing is not always a viable solution.

Concerning the LDHI memory consumption, we have to consider that the PIT is not heavily loaded (see Figure 1.18(b)) since each entry remains in the pending state for a very short period of time, thus the memory usage is definitely negligible. Regarding this last point, we would like to add that the Lifetime set within the Interest packets used by the server to trigger a user request might be in theory nearly zero, that is such packets might pass through the router without leaving any trace since they will not call back Data packets (at least, this is not a strict requirement). Of course, this solution is not feasible since PIT traces are also useful to avoid routing loops. As a consequence, the lifetime may be set to a very small value in order to avoid possible loops and also to minimize the memory usage on CCN devices. The memory consumption showed in Figure 1.18(b) refers to the memory portion occupied by names. In addition, some overhead should be considered since each entry has other information (list of nonces, list of interfaces, etc. . .) hence our computation should be refined with some (hopefully small, depending on the PIT implementation details) space overhead. The memory consumption showed in Figure 1.18(b) confirms that LDHI outperforms COPSS in terms of scalability even at ISP scale, generating a small amount of state information. In fact overloading the central Rendezvous Node with a huge amount of state information could limit the functioning of the entire network. In LDHI, we do not have this issue because the information scattered in all COPSS routers by means of the Subscription Tables, is moved at the edge of the network, in particular on the social network application server, which is in charge of maintaining the user relations (what is commonly called the social graph in social network applications). We believe this is a key point to build a network capable of fully supporting scalable applications.



(a) Incoming Interest/Subscribe packets for COPSS and LDHI at Rome node

(b) Outgoing Subscribe packets for COPSS and LDHI at Rome node

(c) Incoming Publish packets for COPSS at Rome node

(d) Incoming Data packets for LDHI at Rome node

(e) Outgoing Publish packets for COPSS at Rome node

(f) Outgoing Data packets for LDHI at Rome node

Figure 1.16: Simulation results depicting the number of incoming/outgoing packets for the Rome central router in both scenarios (COPSS and LDHI). We selected this particular node since it is the most overloaded one and also because, in the COPSS deployment, we select this network device to work as RN.

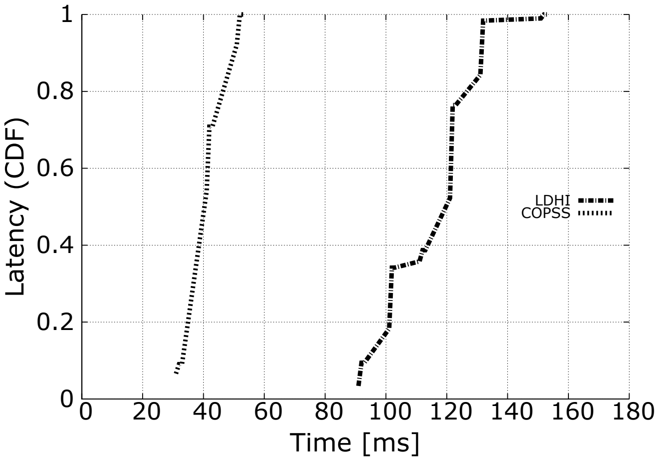
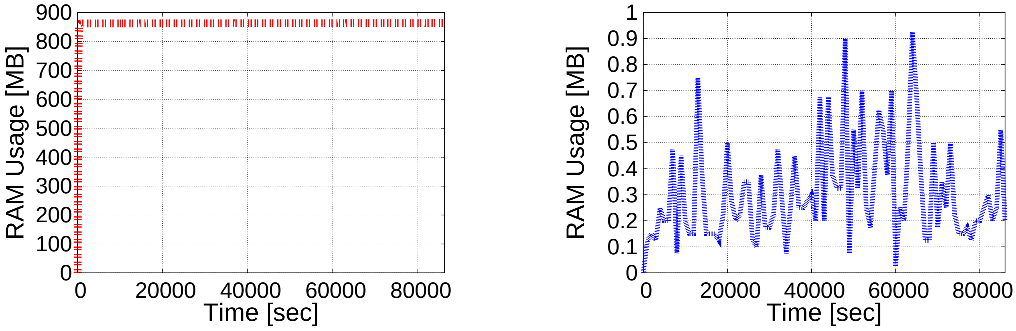


Figure 1.17: Cumulative Distribution Function of the latency experienced by end users in the two proposals



(a) COPSS memory performance at the Rome PoP.

(b) LDHI memory performance at the Rome PoP.

Figure 1.18: Pending Interest Table (LDHI) and Subscription Table (COPSS) comparison.

Chapter 2

Empowering the Internet: SDN and NFV

The architecture and all the mechanisms we described so far have been proposed to optimize content distribution in modern Internet. We sacrificed the simplicity of the IP routing protocol in order to make the network even smarter and more content-oriented. The final goal is that of better supporting the increasing users demand for popular content.

In the networking panorama, however, there is still room for an innovation that is currently hitting an orthogonal target w.r.t content dissemination. We are talking about the architectures and technologies currently known as SDN and NFV. Here the focus is on dynamicity and programmability: the first attempt to introduce such features in the network can be safely connected to the OpenFlow (OF) protocol, whose first version was described in [55] in 2008. Essentially, the idea was to extract the intelligence (the control plane) from forwarding elements and move it in a more centralized network component called Controller. The Controller's logic can be then designed in a more flexible way, i.e. by means of some kind of network oriented programming language. What remains in the forwarding elements according to this new vision, is just a set of simple forwarding rules following the MATCH -> ACTION pattern and these rules are driven by the controller by means of a specific protocol: OF. Since we want to achieve flexibility and agility in the network configuration, the forwarding elements (i.e., the so called OF switches) can programmed *proactively*, that is the controller installs the rules on the switches before the traffic passes through them (e.g., at network bootstrap) or *reactively*, that is the controller installs a rule in a switch when it is required, for example when the switch encounters a packets it has no rule for.

As a consequence of this brief description, it can be inferred that the forwarding process of an OF switch is not based on any traditional L2/L3 protocol (no MAC learning nor IP routing) but rather on this MATCH->ACTION rule pattern. Clearly we can match a variety of fields in a network packet (src/dest port, src/dest

IP, src/dest MAC, . . .) thus forwarding a given flow through a given path thanks to this flexibility introduced by OF rules and the controller logic. Starting from this feature, it is easily feasible to customize the network paths on a per-flow basis which can be useful to make different types of traffic pass through different types of network functions (either physically deployed or virtualized). This opens the view on a new range of services and technologies that can be implemented on practically any existing network with the aim of exposing a new set of functionalities either for network designers and maintainers and also for end users. For example, it is possible to customize the network services offered to a given traffic (user) by providing additional functions (web cache, wan accelerators, security functions, . . .) and this is also known as Service Function Chaining (SFC). In this chapter we will analyze new architectures and solutions proposed in this context to solve different problems. For example we present an efficient data exchange mechanisms to make it possible to move packets from a Virtual Network Function (VNF) to another hosted on the same physical machine. We also propose and validate a solution to check safety properties on dynamically changing networks. Indeed while dynamicity can bring automation and easy-to-use tools to continuously change the network forwarding behaviour, tools to check that networks configuration makes sense and does not break network consistency and integrity must be designed and integrated into such flexible environments. In the next sections we will cover (and solve) all these issues.

2.1 New architectures for SDN

Portions of this section (partially published in [9]) are also part of the PhD thesis of Ivano Cerrato (“High Performance Network Function Virtualization for User-Oriented Services") who collaborated to this work.

2.1.1 Introduction

The already introduced Network Function Virtualization (NFV) [24] proposes to replace dedicated middleboxes, used to deliver a multitude of network services by means of a growing number of (cascading) dedicated appliances, with software images that run on general-purpose servers. This allows leveraging high-volume standard machines (e.g., Intel-based blades) and computing virtualization to consolidate and optimize the processing in the data plane of the network. This results in a more flexible deployment of network applications (e.g., NAT, firewall), in lower capital and operating costs for the hardware thanks to the possibility to deploy many different (small) Virtual Network Functions (VNF) on the same (standard) computer, and in simpler and more reliable networks. In addition, while appliances are often dedicated to a single tenant, servers can be multitenant, hence being able to host services belonging to different actors [70], such as a traffic monitor belonging to the operator and a firewall belonging to a given company, with even more advantages in terms of consolidation.

When several VNFs are executed in the same server, an incoming packet can traverse an arbitrary number of VNFs before leaving the middlebox (i.e., a *function chain*, as shown in Figure 2.1). The exact sequence of functions traversed by a packet can be determined only at run-time, by inspecting the packet. In fact, (*i*) packets belonging to different tenants can traverse different functions, and (*ii*) packets belonging to the same tenant can experience different paths (e.g., when only a portion of traffic needs to undergo a deep packet inspection). Packets can also be modified in transit (e.g., a NAT changes the source IP address), hence requiring that a packet is re-analized when it leaves a VNF, to determine what is next. As depicted in Figure 2.1, this requires that each server includes a module (usually referred as *virtual switch* or *vSwitch*) that classifies each packet to determine which is the next function to traverse and then delivers the packet to it.

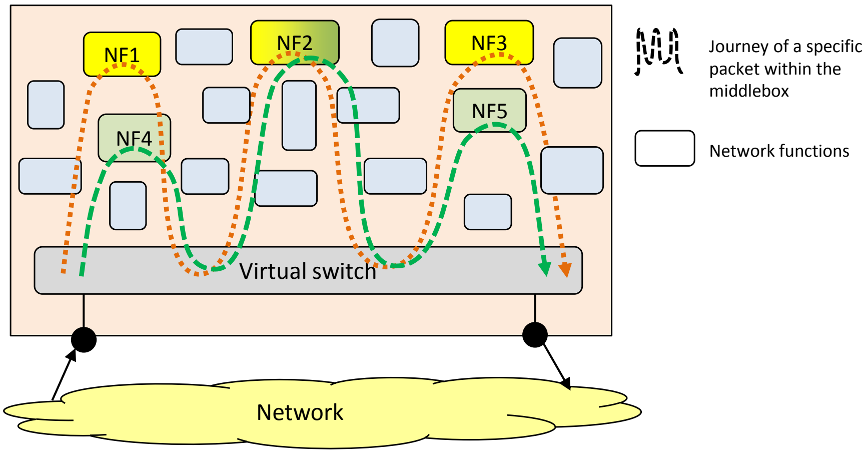


Figure 2.1: Function chains deployed in a middlebox.

Basing on some preliminary work [10], this proposal evaluates a new architecture for moving network packets between different functions, by means of a vSwitch. This solution, which is based on circular lock-free First-In-First-Out (FIFO) buffers managed by ad-hoc algorithms, is designed to:

- *(i)* guarantee *traffic isolation* between functions, so that a function can only access the portion of traffic that is expected to flow through it, hence limiting the potential hazards due to malicious applications and provide an effective support to multitenancy;

- *(ii)* provide excellent *scalability* by allowing to consolidate a huge number of VNFs on the same server;

- *(iii)* achieve high *performance* in terms of data movement speed among different VNFs, similarly to what is required in physical servers among different hardware modules [95].

Scalability and performance are obtained also by taking care of implementation details such as exploiting cache locality as much as possible and limiting the number of context switches, since their costs would introduce an excessive overhead. The correctness of the data exchange algorithms (e.g. absence of concurrency hazards) is guaranteed by means of formal verification.

This section focuses on VNFs that *(i)* implement a pass-through behavior (each packet received is sent again to the network), *(ii)* may drop packets or *(iii)* may generate new packets as a consequence of a packet just received (e.g., an ARP reply as a consequence of an ARP request). This allows us to cover the vast majority of network middleboxes, including for example NATs, firewalls, traffic monitors, and intrusion detection systems.

2.1.2 Related Work

The efficient *lock-free* implementation of FIFO queues has been largely investigated in the past. For instance, [57] and [29] propose lock-free algorithms that operate on FIFO queues managed as non-circular linked-lists. Similar proposals can be found in [67] and [32], which also require to manage a pool of pre-allocated memory slots. However, all the solutions proposed so far are usually based on uni-directional flows of data according to the *producer-consumer* paradigm, which is not an optimal solution for managing the bi-directional data flows occurring in the virtualized environments we are considering. In fact, in these environmnts, a packet always goes from the virtual switch to the VNF and then back to the virtual switch. Using classical uni-directional producer-consumer solutions requires the VNF to remove data just received from a first queue and to write them into a second queue used for sending the data back. This implies that data are always copied once in this trip, which may limit the throughput of the system particularly when several functions have to be traversed (hence several copies have to be carried out).

Another possible way to efficiently exchange data between applications can be seen in the context of a lock-free operating system, in which [52] and [53] present a single producer/consumer and a multi-producer/multi-consumer algorithm to manage circular FIFO queues. A similar idea has been proposed by Intel in the DPDK library and in [82], whose algorithms have been designed to operate in contexts where many processes can concurrently insert items into a *shared* buffer or remove them. However, those proposals are not applicable in our case because they cannot guarantee isolation between VNFs due to the presence of a unique shared buffer. Similar considerations can be made for ClickOS [50, 51] (based on the VALE virtual switch [71]) and NetVM [35], which instead targets network function chains.

ClickOS uses two unidirectional queues with the necessity to copy packets once; NetVM uses two unidirectional queues between "untrusted" functions, while switching to a unique shared buffer (handled in zero-copy) among "trusted" functions, hence impairing traffic isolation requirement. MCRingBuffer [45], instead, defines an algorithm to exchange data between one producer and one consumer running on different CPU cores, which is particularly interesting for its efficient implementation of memory access patterns; in fact, part of those techniques were reused in our implementation as well (Section 2.1.7).

Solutions such as Xen [5], and Hyper-Switch [69] address the problem of efficiently exchanging packets between different entities such as virtual machines (VM) running on the same server, which looks similar to our problem of chaining network functions. However, their architecture is designed for packets that originate or terminate their journey in a VM, not for pass-through functions. This implies different architectural choices such as different buffers for packets flowing in different directions, albeit integrated with a complex data exchange mechanism based on swapping memory pages rather than copying packets between the hypervisor and the VM [5]. It is also worth mentioning that network-aware CPU management techniques have been proposed in the context of Xen for improving the performance of virtual servers hosting these network applications [31].

Virtual switches such as OpenvSwitch (OVS) [64] and the eXtensible Datapath daemon (xDPd) are used to implement network function chains (as shown respectively in [7, 11]), although they appear in some way orthogonal to our proposal. In fact, they implement the classification and forwarding mechanism (either based on the traditional L2 forwarding or on the more powerful Openflow protocol [55]), but do not focus on the data exchange mechanism which is often based on bi-directional FIFO queues (in some case a shared memory can be configured). In this respect, our mechanism can be built on top of those existing solutions to improve their data transfer capabilities, hence the overall performance of the system.

2.1.3 The data exchange architecture

This section describes the proposed architecture, designed to efficiently implement function chaining within a single middlebox. In particular, the section first provides an overview of the architecture and then dives into the details of the packet exchange algorithm.

In our architecture, we define the *Master* as the module that plays the role of the vSwitch, while VNFs are represented by modules called *Workers*. Moreover, a *token* is a generic data unit exchanged between the Master and the Workers. The token represents a packet in the VNF use case, but our mechanism could be used to exchange any kind of data, according to the specific use case implemented.

2.1.4 Operating context

VNFs are pieces of software operating on the data plane of the network that, in the vast majority of cases, forward their packets with minimal (or no) changes, allowing packets to continue their journey toward the final destination. However, some functions (e.g., firewall) may need to drop packets, which should not be sent back to the network after their processing. Other functions, instead, may need to send new packets as a consequence of a previously received packet. For example, a bridging module may need to duplicate a broadcast packet several times (e.g., once for each interface of the middlebox) and then provide all these copies to the next functions in the chain. Similarly, another function may extend a packet (e.g., by adding a new header) so that it exceeds the MTU of the network; this packet must then be fragmented, and all the fragments must be sent out.

Hence, our architecture must take all these requirements into account and must be able to efficiently move all the above traffic within the middlebox in order to allow flexible function chains. As depicted in Section 2.1.1, this requires a fast and efficient mechanism to move data between the vSwitch and the VNFs, which translates into the necessity of a dedicated data dispatching mechanism, being this component one of those that has the biggest impact on the system performance.

2.1.5 Architecture Overview

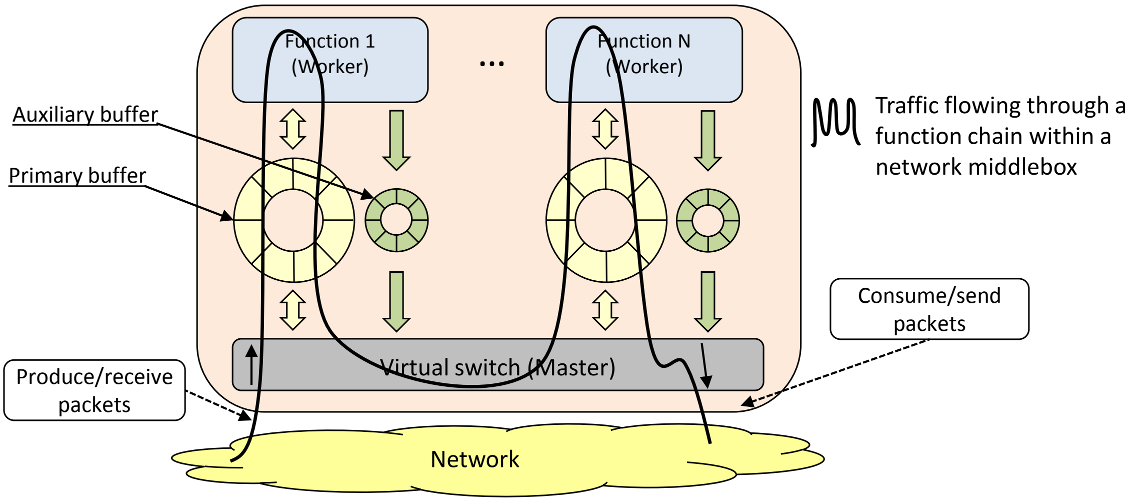


Figure 2.2: Deployment of the algorithm within a middlebox.

As shown in Figure 2.2, our architecture is based on a set of lock-free ring buffers; in particular, the Master shares two buffers with each Worker, which are managed through different (but not independent) parts of the same exchange algorithm.

The *primary buffer* is used to exchange pass-through tokens, i.e., data that go from the Master to the Worker, and back from the Worker to the Master. In particular, the proposed solution has the peculiarity of allowing the Worker to return data back without any copy. Instead, the *auxiliary buffer* is used to support another kind of traffic we envision in our use case scenario, namely Workers that can possibly generate new tokens as a consequence of the data received from the Master, such as an *ARP reply* packet generated in response to an *ARP request*, or when a packet has to be modified but it results in an excess of the MTU, hence requiring the creation of another packet. This second buffer is unidirectional and it is only used by the Worker to provide "new" data to the Master.

Each buffer slot (both primary and auxiliary) includes some flags in addition to the real data, which are used to identify the content of each slot; more details will be presented in the next sections. Finally, buffer slots are currently of fixed length and equal to the network MTU size; however the extension of the algorithm to handle variable slot sizes, tailored to the length of the packet actually received, is trivial.

2.1.6 Execution model

The Master operates in polling mode, i.e., it continuously checks for new tokens and inserts them into the primary buffer shared with the target Worker. This operating mode has been chosen because the middlebox (and then the Master itself) is supposed to be traversed by a huge amount of traffic; hence, a blocking model would be too penalizing because it would require an interrupt-like mechanism to start the Master whenever new data are available. This could significantly drop the performance with high packet rates [58]. In fact, interrupt handling is expensive in modern superscalar processors because they have long pipelines and support out of order and speculative execution [21], which tends to increase the penalty paid by an interrupt.

Vice versa, since the traffic entering into a specific Worker is potentially a small portion compared to the one handled by the Master, a blocking model looks more appropriate for this module. This ensures the possibility to share CPU resources more effectively, which is important in multi-tenant systems where potentially a large number of Workers are active. Hence, when a Worker has no more data to be processed, it suspends itself until the Master wakes it up by means of a shared semaphore.

2.1.7 Basic algorithm: handling pass-through data

The algorithm used to move data from the Master to the Workers (and back) requires the sharing of some variables (underlined in the pseudocode shown in the following), a semaphore, and the primary buffer between the Master and each Worker. In particular, in this section we assume the presence of the Master and a *single* Worker, while its extension to several Workers is trivial.

Algorithm 1 provides the overall behavior of the Master and shows how it cyclically repeats the following three main operations: *(i)* in lines 14-21 it produces new data (line 19), which corresponds to the reception of tokens from the network interface card (NIC) in our case, and immediately provides them to the Worker through the primary buffer (line 20); *(ii)* it reads the tokens already processed by the Worker from the primary buffer (line 22), and finally *(iii)* it wakes up the Worker if there are data waiting for service for a long time in order to avoid starvation (line 23). From lines 14-21, it is evident that the Master produces several tokens consecutively, in order to better exploit cache locality. Furthermore, if the buffer is full (line 15), it stops data production and starts removing the tokens already processed by the Worker from the buffer.

Algorithm 1 Executing the Master

```
 1: Procedure master.do()
 2:
 3: {Initialize shared variables}
 4: M.prodIndex ← 0
 5: W.prodIndex ← 0
 6: workerStatus ← WAIT_FOR_SIGNAL
 7:
 8: {Initialize private variables of the Master}
 9: M.consIndex ← 0
10: timeStamp ← 0
11:
12: {Execute the algorithm}
13: while true do
14:    for i = 0 to (i < N or timeout()) do
15:       if M.prodIndex == (M.consIndex−1) then
16:          {The buffer is full}
17:          break
18:       end if
19:       data ← master.produceData()
20:       master.writeDataIntoBuffer(data)
21:    end for
22:    master.readDataFromBuffer()
23:    master.checkForOldData()
24: end while
```

Algorithm 2 details the mechanism implemented in the Master to send data to the Worker. As shown by line 8, a token is inserted into the slot pointed by the shared index `M.prodIndex` as soon as it is produced; however, the Worker is awakened only if at least a given number of tokens (i.e., `MASTER_PKT_THRESHOLD`) are waiting for service in the primary buffer (lines 10-13). Thanks to this threshold, we avoid to wake up the Worker for each single token that needs to be processed, hence improving performance because (*i*) it reduces the number of context switches and (*ii*) it increases cache locality, for both data and code. Since a token is inserted into the buffer as soon as it is produced regardless of the fact that the Worker is running or not, and since the Worker will suspend itself only when the buffer is empty (as detailed in Algorithm 5), the Worker is able to process a huge amount of data consecutively, thus improving system performance.

Algorithm 2 The Master writing data into the primary buffer

```
 1: Procedure master.writeDataIntoBuffer(Data d)
 2:
 3: if M.prodIndex == M.consIndex then
 4:     {The buffer is empty}
 5:     timeStamp ← now()
 6: end if
 7:
 8: buffer.write(M.prodIndex,d)
 9: M.prodIndex++
10: if buffer.size() > MASTER_PKT_THRESHOLD and
    (workerStatus ≠ SIGNALED) then
11:     workerStatus ← SIGNALED
12:     wakeUpWorker()
13: end if
```

Our algorithm avoids the starvation of tokens sent to a Worker, especially when the system is in underload conditions. This is done by means of a timeout event, which wakes up the worker even if the abovementioned threshold is not reached yet. In particular, the Master acquires and stores the current time whenever it inserts a new token and the buffer is empty (lines 3-6 of Algorithm 2). This way, the Master knows the age of the oldest token and it is able to possibly wake up the Worker also depending on the value of a given time threshold, as shown in Algorithm 3.

Algorithm 3 The Master waking up the Worker due to a timeout

```
1: Procedure master.checkForOldData()
2:
3: if buffer.size() > 0 and (workerStatus ≠ SIGNALED) and
   ((now() − timeStamp) > TS_THRESHOLD) then
4:     workerStatus ← SIGNALED
5:     wakeUpWorker()
6: end if
```

The functions described in Algorithm 2 and Algorithm 3 need to know whether the Worker is already running or not in order to avoid useless Worker awakenings. This information is carried by the shared variable `workerStatus`, which is set to `SIGNALED` by the Master just before waking up the Worker (line 11 of Algorithm 2 and line 4 of Algorithm 3), and changed to `WAIT_FOR_SIGNAL` by the Worker just before suspending itself (line 22 of Algorithm 5). This way, the Master can test this shared variable to have an indication about the Worker status, and then wake it up only when necessary.

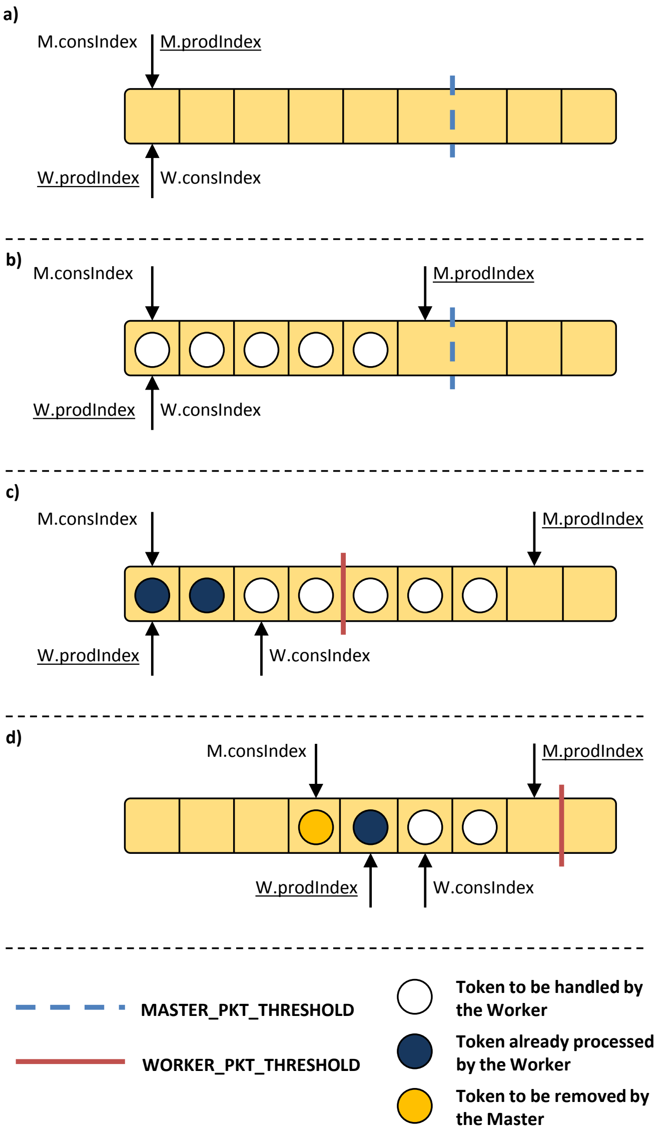


Figure 2.3: Run-time behavior and indexes of the algorithm.

Algorithm 4 shows how the Master removes the data that have already been processed by the Worker. In particular, it consumes all the tokens until the index `M.consIndex` does not reach the index `W.prodIndex`, incremented by the Worker each time it has handled a batch of tokens, as detailed in Algorithm 5. In this way, also the Master reads several consecutive data from the primary buffer in order to better exploit cache locality.

Algorithm 4 The Master reading data from the primary buffer

```
 1: Procedure master.readDataFromBuffer()
 2:
 3: if buffer.size() then
 4:    if M.consIndex $\neq$ W.prodIndex then
 5:       timeStamp $\leftarrow$ now()
 6:       while M.consIndex $\neq$ W.prodIndex do
 7:          if not buffer.dropped(M.consIndex) then
 8:             master.consumeData(buffer.read(M.consIndex))
 9:          end if
10:          M.consIndex++
11:       end while
12:    end if
13: end if
```

Notice that Algorithm 4 also considers those tokens provided by the Master to the Worker, and dropped by the Worker itself. In case of dropped data, the Master receives back an empty slot, identified through the flag `dropped`. The content of a slot is only consumed if this flag is zero, otherwise the Master just increments the `M.consIndex` and moves on to the next slot of the buffer, as shown in lines 7-10. This prevents the Master from reading a slot with a meaningless content.

Algorithm 5 details the operations of the Worker. As evident from lines 12-23, whenever a Worker wakes up, it processes all the tokens available in the primary buffer (i.e., all the slots of the buffer with indexes less than `M.prodIndex`). Only at this point (line 24), as well as after it has processed a given amount of data (lines 13-16), the Worker updates the shared index `W.prodIndex`, so that the Master can consume all the tokens already processed by the Worker itself. This way, the Master will be notified for data availability only when a given amount of tokens are ready to be consumed, with a positive impact on performance. It is worth noting that this batching mechanism is different from the one implemented when the Master sends data to the Worker. In fact, in that case, the Worker is woken up when the amount of data into the buffer is higher than a threshold, while the `M.prodIndex`, used by the Worker to understand when it has to suspend itself, is incremented each time a new data is inserted. Here, instead, the `W.prodIndex` (i.e., the index used by the Master to know when the consuming of tokens must be stopped) is not updated each time the Worker processes a data. As a consequence, it is possible that some tokens have already been processed by the Worker, but it has still to update the `W.prodIndex` and then the Master cannot consume them in the current execution of Algorithm 4. This results in a slightly higher latency for these tokens, but in better performance for the system thanks to this batching processing enabled into the Master. As a final remark, lines 18-20 show that the Worker can drop the token under processing by setting the `dropped` flag in the current slot of the primary buffer.

Algorithm 5 Executing the Worker

```
 1: **Procedure** worker.do()
 2:
 3: {Initialize private variables of the Worker}
 4: W.consIndex ← 0
 5: pkts_processed ← 0
 6:
 7: {Execute the algorithm}
 8: **while** true **do**
 9:     waitForWakeUp()
10:     W.consIndex ← $\underline{\text{W.prodIndex}}$
11:     pkts_processed ← 0
12:     **while** W.consIndex ≠ $\underline{\text{M.prodIndex}}$ **do**
13:         **if** pkts_processed == WORKER_PKT_THRESHOLD **then**
14:             pkts_processed ← 0
15:             $\underline{\text{W.prodIndex}}$ ← W.consIndex
16:         **end if**
17:         toBeDropped ← $\underline{\text{buffer}}$.process(W.consIndex)
18:         **if** toBeDropped **then**
19:             $\underline{\text{buffer}}$.setDropped(W.consIndex)
20:         **end if**
21:         W.consIndex++
22:         pkts_processed++
23:     **end while**
24:     $\underline{\text{W.prodIndex}}$ ← W.consIndex
25:     $\underline{\text{workerStatus}}$ ← WAIT_FOR_SIGNAL
26: **end while**
```

Figure 2.3 depicts the status of the primary buffer[1] and the indexes used by the algorithm in four different time instants. In Figure 2.3(a) the buffer is empty, and then all the indexes point to the same position. Instead, in Figure 2.3(b) the

[1] For the sake of clarity, the figure represents the shared buffer as an array instead of a circular FIFO queue.

Master has already inserted some data into the buffer, but the Worker is still waiting since the `MASTER_PKT_THRESHOLD` has not been reached yet. Figure 2.3(c) depicts the situation in which the Master has woken up the Worker, which has already processed two items. Notice that, since the `WORKER_PKT_THRESHOLD` has not been reached yet, the `W.prodIndex` still points to the oldest token in the buffer. Instead, in Figure 2.3(d) this threshold is passed and the Master has already consumed some data.

Extended algorithm: handling Worker-generated data

Our architecture handles also Workers that may need to generate *new* data as a consequence of the token just received from the Master but, as evident, this cannot be done with the primary buffer alone as Workers cannot *inject* new data into the primary buffer. In fact, the Worker can just modify (potentially completely) pass-through tokens, i.e., data received from the Master that must be sent back to the Master itself or, at most, it can drop these tokens.

Since network applications forward most of the packets without performing any manipulation on it, we decided to keep the primary buffer as simple as possible for the sake of speed, while adding a new lock-free ring buffer, i.e., the *auxiliary buffer*, to handle the case in which new data have to be provided to the Master. This buffer, in which the Worker acts as the producer while the Master plays the role of the consumer, is managed through two indexes; moreover, it requires a further flag in each slot of the primary buffer, which indicates whether the next token should be read from the primary or the auxiliary buffer.

Algorithm 6 The Worker writing *new* data into the auxiliary buffer

```
Procedure worker.writeDataIntoAuxBuffer(Data[] newData, Index W.consIndex)

while data ← newData.next() do
    if auxProdIndex == (auxConsIndex-1) then
        {The auxiliary buffer is full}
        break
    end if
    auxBuffer.write(auxProdIndex,data)
    auxBuffer.setNext(auxProdIndex)
    auxProdIndex++
end while
auxBuffer.resetNext(auxProdIndex-1)
buffer.setAux(W.consIndex)
```

Algorithm 6 details how the Worker sends new data to the Master, as a consequence of the processing of the token at position `W.consIndex` in the primary buffer.

As shown in lines 3-11, several data can be generated for a single token received from the Master, which are all linked to the same slot of the primary buffer. A first flag, called `aux`, is set in the slot of the primary buffer to signal that the next slot to read is the one on top of the auxiliary buffer (line 13). Instead, the `next` flag set in a slot of the auxiliary buffer tells that the next packet has still to be read from the auxiliary buffer, instead of returning to the next slot of the primary buffer.

The reading procedure is described in Algorithm 7. When the Master encounters a slot with the `aux` flag set in the primary buffer, it processes a number of tokens in the auxiliary buffer, starting from the slot pointed by `auxConsIndex` until the `next` flag is set. Moreover, according to lines 4-7 of Algorithm 6, if the auxBuffer is full, new tokens that the Worker may want to send to the Master are dropped.

It is worth noting that the `auxConsIndex` is only moved by the Master and, at the beginning of Algorithm 7, it points to the first data to be read in the auxiliary buffer while, at the end of the pseudocode, it points to the first slot will be read the next time the Master will process data in the auxiliary buffer.

Algorithm 7 The Master reading data from the auxiliary buffer

```
Procedure master.readDataFromAuxBuffer()

while true do
    master.consumeData(auxBuffer.read(auxConsIndex))
    if not auxBuffer.next(auxConsIndex) then
        auxConsIndex++
        break
    end if
    auxConsIndex++
end while
```

Figure 2.4 depicts the primary buffer with some slots linked to the auxiliary buffer. In particular, the slot pointed by `M.consIndex` is associated with two data of the auxiliary buffer, i.e., the one pointed by `auxConsIndex` and the following one, which has the `next` flag reset to indicate that the next slot is not linked with the current slot in the primary buffer. Instead, the next token in the primary buffer is not associated with the secondary buffer (the `aux` flag is reset), while the third slot contains data dropped by the Worker; despite this, the slot is linked to three data in the auxiliary buffer. In other words, the configuration in which `aux == 1` and `dropped == 1` is valid and it enables to completely replace a packet with a new one.

Implementation

Since the achievable performance depends not only on design but also on implementation issues, this section presents some implementation choices that can improve

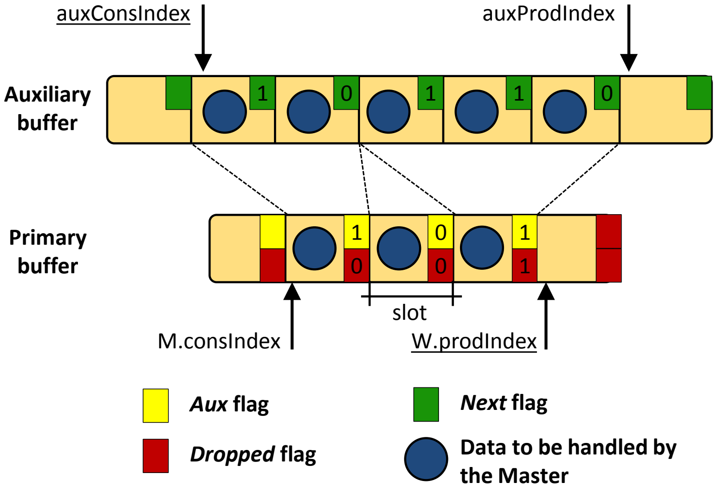


Figure 2.4: Binding primary buffer - auxiliary buffer.

performance and scalability and that have been adopted in our ptototype implementation.

Private copies of shared variables. As in many algorithms derived from the producer-consumer problem, also in our case we need to keep two processes in sync by means of a pair of shared variables, one written only by the first process, the other one written only by the second process. Although in this case concurrency issues are limited (no contention can occur because the two processes never try to write the same variable at the same time), the actual implementation on real hardware can introduce additional issues, as shown in MCRingBuffer [45]. In fact, when a first CPU core modifies the content of a variable that is shared with a different CPU core, the entire cache line (64 bytes long on the modern Intel architectures) of the second core containing that variable is invalidated. If the second core needs to read that variable, the hardware has to retrieve this value either from the shared cache (e.g., the L3 in many recent Intel architectures) or from the main memory, with a consequent performance penalty. In our algorithm, this problem occurs for `M.prodIndex`, incremented by the Master whenever a new token is inserted into the primary buffer and read by the Worker, and for `W.prodIndex`, incremented by the Worker and read by the Master. However, our algorithm is robust enough to operate correctly even if those variables are not perfectly aligned. As a consequence, the Worker creates a private copy of `M.prodIndex` just after waking up, while the Master copies in a private variable the content of `W.prodIndex` before reading data from the shared buffer.

Shared variables on different cache lines. Because of the same problem mentioned in the previous paragraph (a CPU core can invalidate an entire line of cache in the other cores), our code implements a cache line separation mechanism (similar to MCRingBuffer [45]), which consists in storing each shared variable (possibly extended with padding bytes) on a different cache line. This way, when the Master changes, for instance, the value of `prodIndex`, the cache line containing `workerIndex` is not invalidated in the private cache of the core where the Worker is executed.

Alignment with cache lines. In case of a cache miss, the hardware introduces a noticeable latency because of the necessity to transfer the data from the memory to the cache, which happens in blocks of fixed size (the *cache line*). From that moment, all the memory accesses within that block of addresses are very fast, as data are served from the L1 cache. In order to minimize the number of cache misses (and the associated performance penalty), our prototype was engineered to align the most frequently accessed data so that they span across the minimum set of cache lines. In particular, the starting memory address of the packet buffers and their slot sizes are multiple of the cache line size; the same technique is used for minimizing the time for accessing the most important data used in the prototype.

Use of huge memory pages. Huge pages are convenient when a large amount of memory is needed because they decrease the pressure on the Translation Lookaside Buffer (TLB) for two reasons. First, the load of virtual-to-real address translation is split across two TLBs (one for huge pages and the other for normal memory), preventing normal applications (based on normal pages) from interfering with the packet exchange mechanism (which uses huge pages). Second, they reduce the number of entries in the TLB when a large amount of memory is needed. We use the huge pages for the shared (primary and auxiliary) buffers; the drawback is the potential increase of the total memory required by the algorithm because the minimum size of each buffer increases from 4KB to 2MB.

Preallocated memory. Dynamic memory allocation should be avoided during the actual packet processing, as this would heavily decrease the performance of the whole system. In our case, all the buffers used by the packet exchange mechanisms are allocated at the startup of each Worker, allowing the system to add/remove workers at run-time while at the same time avoiding dynamic memory allocation.

Emulated timestamp. Getting the current time is usually rather expensive on standard workstations as it requires the intervention of the operating system and, often, an I/O operation involving the hardware clock. In our case we emulate the timestamp, which is needed to wake up a Worker when packets are waiting for service for too long time, by introducing the concept of *current round*, that is the number of loops executed by the Master in Algorithm 1. As a consequence, our implementation schedules a Worker for service when there are packets waiting for more than N rounds; this number can be tuned at run-time based on the expected load on the Master.

Batch processing. Batch processing is convenient because it keeps a high degree of code locality, with a positive impact on cache misses. Our prototype implements batch processing whenever possible, e.g., the Master reads all waiting packets from a worker before serving the next, and Workers process all the packets in their queue before suspending themselves; the drawback is the potential increase of the latency in the data transfer.

Semaphores. A simple POSIX semaphore is used to wake up a Worker that has data waiting to be processed (i.e., at least `MASTER_PACKET_THRESHOLD` packets are queued, or some packets are waiting for long time and the timeout expires). Although POSIX semaphores are implemented in kernel space, their impact is acceptable as they are rarely accessed by algorithm design. Instead, no explicit signal is used in the other direction: the shared variables `M.consIndex` and `W.prodIndex` are used by the Master to detect the presence of packets than needs to be read from the buffer.

Threading model. Context switching should be avoided whenever possible because of its cost, particularly when this event happens frequently (such as in packet processing applications, which are usually rather simple and often handle a few packets in a row). For this reason, the Master is a single thread process, cycling on a busy-waiting loop and consuming an entire CPU core, while Workers (which are single-thread processes as well) work in interrupt mode and share the remaining CPU cores. While the Master can be simply parallelized over multiple cores as long as the function chains are not interleaved[2], by design our implementation keeps it locked to a single core as we would like to allocate the most part of the processing power to the (huge number of) Workers, which will host the network functions that are in charge of the actual (useful, from the perspective of the end users) processing.

2.1.8 Formal verification

Assessing the correctness of an algorithm is often not straightforward, hence we built an abstract model of the Master with a single Worker in order to formally check some fundamental properties. We do not consider a plurality of Workers because the interaction between the Master and a Worker is independent of the interaction with any other Worker, hence this approach is sufficient to demonstrate the correctness of the whole system. In particular, we only focus on the primary buffer as its operation is one of the main contributions of our work and hence it needs a proof of correctness. The auxiliary buffer, instead, is not explicitly verified as it is managed as a standard producer/consumer system, which has been already studied and validated in the existing literature.

The model of our algorithm has been developed in Promela [33], a well known

[2] Interleaved chains may introduce additional complexity because multiple masters may collide when feeding a single Worker; this would require an extension of our algorithm (no longer lock-free) that is left to a future work.

modeling language that, in conjunction with the SPIN [34] model checker generator, can be used to formally verify distributed and concurrent software against certain desired properties. The main purpose of the model checking technique is to explore all the possible states of a system and verify whether the specified properties hold in *each* execution path. Whenever the model checker finds an execution path that leads to a property violation, it provides the full counter-example with all the steps needed to reach the undesired behavior.

When creating an accurate model of the system, it is very important to keep the nature of the problem *tractable*, as model checking verification tools tend to exploit a massive amount of memory (state-space explosion problem). Therefore, the actual model of the data exchange mechanism has been built by omitting some implementation details that are not relevant for the analyzed properties in order to reduce the overall number of states. This is possible because many system states (or runs) are mapped to the same abstract state (or run). A more detailed description of our model will be provided in Section 2.1.8.

Properties specification

Given the structure of our algorithm, we can identify six properties that *must* be always satisfied. The first two properties refer to conditions on some key variables that must be verified to guarantee that no slots will be erroneously overwritten, formally defining regions of the buffer that are "owned" by one of the two modules (the Master and the Worker) at a given time.

Property 1. *`W.prodIndex` must never exceed `M.prodIndex`.*

`M.prodIndex` indicates the first empty position in the primary buffer that must be fulfilled by the Master. Hence, it represents a boundary that the Worker must never pass.

Property 2. *`M.consIndex` must never exceed `W.prodIndex`.*

`M.consIndex` represents the position of the token being processed by the Worker, while `W.prodIndex` identifies the position of the last "processable" token (for the Worker).

We also consider two additional safety properties, which must be satisfied by the system. Specifically we require that:

Property 3. *The number of pending tokens delivered by the Master to the Worker and not yet processed by the Worker itself is, at any time, a non negative integer not exceeding the maximum number of elements that the buffer can store, namely (N - 1), where N is the total buffer size:*

$$0 <= tokens\_master\_to\_worker <= (N-1)$$

Property 4. *The number of pending tokens delivered by the Worker to the Master and not yet processed by the Master itself is, at any time, a non negative integer not exceeding the maximum number of elements that the buffer can store, namely (N - 1), where N is the total buffer size:*

$$0 <= tokens\_worker\_to\_master <= (N - 1)$$

Our circular buffer implementation always leaves at least one empty position, so as to distinguish the cases where the buffer is completely full or completely empty. This is why the actual buffer capacity is N-1.

Finally, we consider two more properties related to the overall system behavior.

Property 5. *Deadlock absence.*

This property is automatically checked by SPIN, and in our case it means that neither the Master nor the Worker ever enter an infinite waiting situation.

Property 6. *Livelock absence.*

This last property ensures that some useful work is eventually done by the Master. Here the notion of "useful work" is intended as the Master capability to produce, sooner or later, new tokens for the Worker, e.g., by inserting new data into the buffer. This is an important property verified under the assumption that a fairness constraint exists during the verification phase, i.e., we assume the process scheduler gives the possibility to both the Master and the Worker to execute, sooner or later, some instructions. This is a reasonable hypothesis since most modern execution environments implement scheduling algorithms to avoid process starvation. Please refer to the full model code for further specifications details (references below).

Model details

The primary buffer

Our abstract model does not require the modeling of realistic data into the buffer but only the buffer status; hence, only indexes are modeled. Another parameter that is crucial for the model is the buffer size, meaning the actual number of tokens that can be stored into the buffer.

The semaphore and the functions implementation

The model of the semaphore consists in an asynchronous channel shared between the master and the worker. Basically, the *blocking wait* operation corresponds to reading a packet from the channel, while the *signaling* operation is modeled by writing a packet into the channel. This is a very common pattern, useful to implement various kinds of communication/synchronization primitives between two or more entities.

The functions presented in the pseudocode in Section 2.1.3 are modeled as Promela processes since the language does not provide an explicit way to represent functions and their returned value. We exploit the following pattern: the caller sends a token through a synchronous channel shared with the callee in order to pass the control to the invoked process. Then, it performs a *receive* operation on the same channel in order to be awakened from the other end-point when the processing has terminated. Notice that the channel can also be used to pass arguments to and from the called process/function.

The Master and the Worker

The two main entities, the Master and the Worker, are modeled as two independent, concurrently running processes. They share the `M.prodIndex` and `W.prodIndex` variables, and the channel/semaphore (as stated in our pseudo-code in Section 2.1.7). We perform the algorithm verification with a single Worker because a multi-worker scenario does not introduce any possible error, given the independence of the algorithm execution between the Master and each Worker. In other words, for each Worker, the Master employs an independent instance of our algorithm. In order to decrease the amount of states to be verified by the model checker, and hence reduce the overall verification time to a reasonable value, we use the following abstractions: *(i)* the if-statement of Algorithm 3 excludes the check on the timestamp value as the whole model does not contain any explicit information about the elapsing time; *(ii)* the `timeout()` function that is present in the loop guard (Algorithm 1) is replaced by a non-deterministic choice (i.e., rather than modeling a realistic mechanism to implement a timeout event, we instructed the model checker to extract a random value to decide if a timeout has occurred or not). Both these abstractions provide a significant performance enhancement without any loss in terms of exhaustiveness of the verification.

	Parameters			**VERIF. RESULT**
	BUFFER SIZE	**MASTER THRSHLD**	**WORKER THRSHLD**	
Property 1	[1,8]	[1,8]	[1,8]	SUCCESS
Property 2	[1,8]	[1,8]	[1,8]	SUCCESS
Property 3	[1,8]	[1,8]	[1,8]	SUCCESS
Property 4	[1,8]	[1,8]	[1,8]	SUCCESS
Property 5	[1,8]	[1,8]	[1,8]	SUCCESS
Property 6	[2,8]	[1,8]	[1,8]	SUCCESS

Table 2.1: Algorithm verification

Verification results

The model explained above can be exhaustively verified for different values of the main model parameters, as shown in Table 2.1. For each property, the table specifies the considered range of values for the buffer size, the `MASTER_PKT_THRESHOLD` and the `WORKER_PKT_THRESHOLD`. For the sake of scalability of the verification process and without losing in generality, we used rather small values compared to a realistic buffer, which could contain millions of tokens. In fact, possible structural bugs would be detected also in a small size system deployment.

Some inconsistent parameters settings in the considered ranges, such as a threshold greater than the buffer size, are skipped in our verification work. Notice also that, with a buffer size equal to one token, Property 6 is not considered as the buffer cannot contain any token and therefore the master is not able to perform any useful work, according to our definition. Properties 1-5 are verified even without forcing any fairness criterion between the execution of the Worker and the Master, since their satisfaction does not necessarily depend on a particular sequence of processes scheduling.

In conclusion, the results of our verification process completely demonstrate the correctness of the algorithm from different points of view (absence of indexes misbehavior or accidental packets overwriting, and absence of deadlocks and livelocks). The complete Promela code is publicly available [85] and is reported in the following:

Listing 2.1: Algorithm model

```
/* Shared Buffer Algorithm - Code for safety checks is deactivated (commented) -> Only liveness
     properties */

#define WAIT_FOR_SIGNAL                   0                 // the worker is sleeping
#define SIGNALED                                    1                 // the worker is running
#define N 9                                                           // buffer size
#define MASTER_PKT_THRESHOLD 9
#define WORKER_PKT_THRESHOLD 9
#define TIMEOUT                                     1
#define CONTINUE                                    0
#define isFull() (buffer_size == N-1)
#define cond1 (scheduling == 0)
#define cond2 (scheduling == 1)
#define cond3 (old_flag == 0)
#define cond4 (old_flag == 1)
#define cond5 (master_progress == 0)
#define cond6 (master_progress == 1)

/* Boolean variables representing the signs of the inequalities modelling the properties we want
      to check */
//bit    work_index_M_prod_index_inequality;       // W_prod_index < M_prod_index (in the initial
     state)
//bit    M_cons_index_work_index_inequality;       // consumer_index < W_prod_index (in the initial
      state)

/* Packets counters (used to check properties) */
//byte   from_master_to_worker_pkt_counter;        // counter for packets read from the master but
     not yet processed by the worker
//byte   from_worker_to_master_pkt_counter;        // counter for packets processed by the worker
     but not yet taken by the master

/* Variables shared between the master and the worker */
byte     M_prod_index;              // assuming buffer size is less than 255
byte     W_prod_index;              // ...idem
bit      worker_status;             // only two possible states for the worker: SIGNALED |
     WAIT_FOR_SIGNAL
byte     buffer_size;               // actual buffer_size
bit      scheduling = 0;
bit      master_progress = 0;
bit      old_flag = 0;

ltl progress_property
{
        (
                always
                (
                        //// PREMISE: some constraints that we want to impose on non
                             deterministic choices
                        (
                                always( (scheduling == 0) implies (eventually (scheduling == 1))
                                            )       // If I schedule the timeout, I will also
                                     write a packet sooner or later
                        )
                        &&
                        (
                                always( (old_flag == 0) implies (eventually (old_flag == 1))
                                     )       // If packets are not old, they will become so
                                     sooner or later
                        )
                        ////
                )
        )

        implies
        (
                //// CONSEQUENCE: We want to verify the master is performing some progresses
                always( (master_progress == 0) implies (eventually (master_progress == 1))
                             )
                ////
        )

}

/* Channel used to implement the synchronization between master and worker */
chan sem = [2] of {bit};

/* Channels used to realize the communication between the master and various functions */
chan writeDataIntoBuffer_ch = [0] of {bit};
chan checkForOldData_ch         = [0] of {bit};
chan readDataFromBuffer_ch      = [0] of {byte};        // The channel is used to pass (and to
     return) M_cons_index

active proctype master()
{
        /* Shared variables initialization */
        M_prod_index    = 0;
        W_prod_index    = 0;
        worker_status   = WAIT_FOR_SIGNAL;
        buffer_size     = 0;

        /* Private variables */
        byte    i;
        byte    M_cons_index;
        byte    temp_index;
        bool    f_result;
        byte    temp;
        M_cons_index    = 0;

        /*
        d_step {
                from_master_to_worker_pkt_counter = 0;
                from_worker_to_master_pkt_counter = 0;

                work_index_M_prod_index_inequality = 0;
                M_cons_index_work_index_inequality = 0;
        }
        */

        main_loop:
                i = 0;
                inner_loop:
                        master_progress = 0;
                        if
                        :: (i < N);
                                /* Is Timeout expired? */
                                if
                                :: (1 == 1);
                                        scheduling = 1; // No, so continue trying to produce a
                                                packet
                                :: (1 == 1);
                                        scheduling = 0;
                                        goto fine_for;  // Yes, so exit from the loop
                                fi;
                                atomic
                {
                    if
                    :: (M_cons_index == 0);
                        temp_index = N-1;
                    :: else
                        temp_index = M_cons_index-1;
                    fi;
                    temp = M_prod_index;
                }
                if
                :: (temp == temp_index);          // Buffer Full
                    goto fine_for;
                :: else
                    skip;
                fi;
                /* Write data into the buffer */
                writeDataIntoBuffer_ch!1;
                writeDataIntoBuffer_ch?f_result;
            :: else
                goto fine_for;
            fi;
            i = i + 1;
        goto inner_loop;
        fine_for:

        /* Read data from the buffer */
        readDataFromBuffer_ch!M_cons_index;
        readDataFromBuffer_ch?M_cons_index;

        /* Check for old Data */
        checkForOldData_ch!1;
        checkForOldData_ch?_;

    goto main_loop;
}

active proctype worker()
{
    /* Private variables */
    int     pktCnt = 0;
    byte    W_cons_index = 0;
    byte    temp;

    worker_loop:
        sem?_;
        W_cons_index = W_prod_index;
        pktCnt = 0;
        worker_inner_loop:
            temp = M_prod_index;
            if
            :: (W_cons_index != temp);
                if
                :: (pktCnt == WORKER_PKT_THRESHOLD);
                    pktCnt = 0;
                    // Here we must verify if the W_prod_index has reverted
                         to the beginning
                    atomic {
                        /*
                        if
                        :: (W_prod_index > W_cons_index);
                        d_step {
                            M_cons_index_work_index_inequality = 1-
                                 M_cons_index_work_index_inequality;
                            work_index_M_prod_index_inequality = 1-
                                 work_index_M_prod_index_inequality;
                        }
                        :: else skip;
                        fi;
                        */
                        W_prod_index = W_cons_index;
                        /* Check assertions */
                        /*
                        if
                        :: (work_index_M_prod_index_inequality == 0);
                             assert (W_prod_index <= M_prod_index);
                        :: (work_index_M_prod_index_inequality == 1);
                             assert (W_prod_index >= M_prod_index);
                        fi;
                        */
                        /******************/
                    }
                :: else skip;
                fi;
                // process packet: NULL processing in this case
                // skip;
                //progress2: skip;
                d_step {
                    /*
                     * Updating W_cons_index:
                     * Atomic execution (increment + modulo) is OK since
                          W_cons_index is a private
                     * variable of the worker
                     */
                    W_cons_index = (W_cons_index+1) % N;
                    /*
                    from_master_to_worker_pkt_counter--;
                    from_worker_to_master_pkt_counter++;
                    assert(from_master_to_worker_pkt_counter >= 0);
                    assert(from_master_to_worker_pkt_counter < N);
                    */
                }
                pktCnt++;
            :: else goto fine_worker_inner_loop;
            fi;
        goto worker_inner_loop;


        fine_worker_inner_loop:
        // Here we must verify if the W_prod_index has reverted to the beginning
        atomic {
            /*
            if
            :: (W_prod_index > W_cons_index);
                M_cons_index_work_index_inequality = 1-
                     M_cons_index_work_index_inequality;
                work_index_M_prod_index_inequality = 1-
                     work_index_M_prod_index_inequality;
            :: else
                goto update_index;
            fi;
            update_index:
            */
            W_prod_index = W_cons_index;
        }
        worker_status = WAIT_FOR_SIGNAL;
    goto worker_loop;
}

active proctype writeDataIntoBuffer()
{
    byte    temp;
    byte    result;
    bit             status;

    writeDataIntoBuffer_loop:
        writeDataIntoBuffer_ch?_;
        // Write the packet into the buffer
        /* A valid condition for livelock absence check is:
         * more packets will always be written in the buffer
         */
         d_step{
            buffer_size++;
            master_progress = 1;     // We have written a packet into the buffer so
                 we have done useful work
            /*
            from_master_to_worker_pkt_counter++;
            assert(from_master_to_worker_pkt_counter >= 0);
            assert(from_master_to_worker_pkt_counter < N);
            */
        }
        /* Updating M_prod_index */
        temp = M_prod_index;
        temp++;
        if
        :: (temp == N);
            atomic {
                M_prod_index = 0;
                //work_index_M_prod_index_inequality = 1-
                     work_index_M_prod_index_inequality;
                /* Check assertions */
                /*
                if
                :: (work_index_M_prod_index_inequality == 0); assert (
                     W_prod_index <= M_prod_index);
                :: (work_index_M_prod_index_inequality == 1); assert (
                     W_prod_index >= M_prod_index);
                fi;
                */
                /******************/
            }
```

```
                :: else
                        atomic {
                                M_prod_index = temp;
                                /* Check assertions */
                                /*
                                if
                                :: (work_index_M_prod_index_inequality == 0); assert (
                                    W_prod_index <= M_prod_index);
                                :: (work_index_M_prod_index_inequality == 1); assert (
                                    W_prod_index >= M_prod_index);
                                fi;
                                */
                                /******************/
                        }
                fi;
                /* Check whether to wake up the worker or not */
                if
                :: (buffer_size > MASTER_PKT_THRESHOLD);
                        status = worker_status;
                        if
                        :: (status != SIGNALED);
                                worker_status = SIGNALED;
                                sem!1;
                        :: else
                                skip;
                        fi;
                :: else
                        skip;
                fi;
                result = 1;

                fine_writeDataIntoBuffer:
                writeDataIntoBuffer_ch!result;
        goto writeDataIntoBuffer_loop;
}

active proctype checkForOldData()
{
        bit     status;

        checkForOldData_loop:
                checkForOldData_ch?_;
                /* Non-deterministic choice: are packets too old? */
                if
                :: (buffer_size > 0);
                        status = worker_status;
                        if
                        :: (status != SIGNALED);
                                if      // Are packets too old?
                                :: (1 == 1);    // YES
                                        old_flag = 1;
                                        worker_status = SIGNALED;
                                        sem!1;
                                :: (1 == 1);    // NO
                                        old_flag = 0;
                                fi;
                        :: else skip;
                        fi;
                :: else skip;
                fi;
                checkForOldData_ch!1;
        goto checkForOldData_loop;
}

/*
 * The function takes M_cons_index as input argument (through the channel)
 * and returns M_cons_index
 */
active proctype readDataFromBuffer()
{
        byte    M_cons_index;
        byte    temp;

        readDataFromBuffer_loop:
                readDataFromBuffer_ch?M_cons_index;
                readDataFromBuffer_while:
                        /* Read data from buffer */
                        temp = W_prod_index;
                        if
                        :: (M_cons_index != temp);
                                // Consume the packet by decrementing buffer_size (variable used
                                    only by the master)
                                buffer_size--;
                                atomic {
                                        /* M_cons_index is a master's private variable */
```

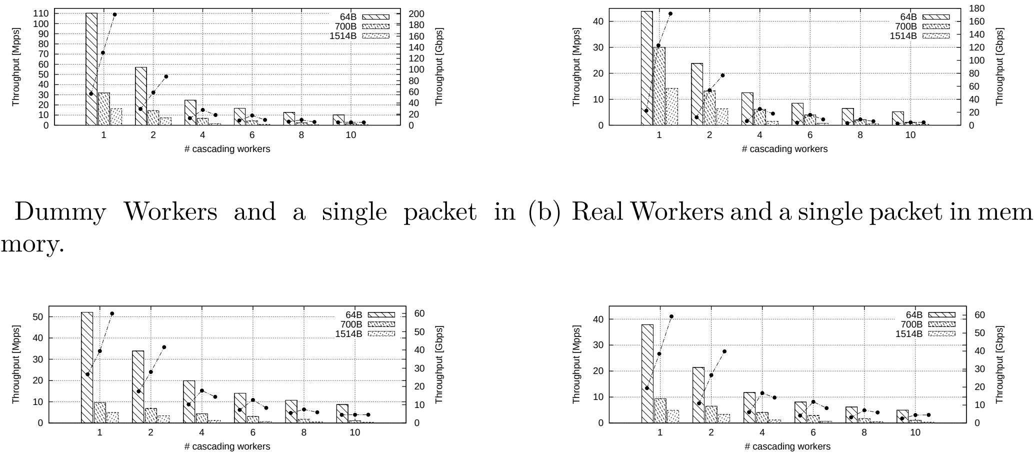
```
                    M_cons_index++;
                    //from_worker_to_master_pkt_counter--;
                    if
                    :: (M_cons_index == N);
                        M_cons_index = 0;
                        //M_cons_index_work_index_inequality = 1-
                          M_cons_index_work_index_inequality;
                    :: else
                        goto readDataFromBuffer_while;
                    fi;
                    /*
                    assert(from_worker_to_master_pkt_counter >= 0);
                    assert(from_worker_to_master_pkt_counter < N);
                    */
                    /* Check assertion */
                    /*
                    if
                    :: (M_cons_index_work_index_inequality == 0); assert (
                      M_cons_index <= W_prod_index);
                    :: (M_cons_index_work_index_inequality == 1); assert (
                      M_cons_index >= W_prod_index);
                    fi;
                    */
                    /******************/
                }
            :: else goto fine_reading_data;
            fi;
        goto readDataFromBuffer_while;
        fine_reading_data:
        readDataFromBuffer_ch!M_cons_index;
    goto readDataFromBuffer_loop;
}
```

2.1.9 Experimental results

In order to evaluate performance and scalability (using the implementation choices described in the previous section), we carried out several tests on our prototype implementation running on a workstation equipped with an Intel i7-3770 @ 3.40 GHz (four CPU cores plus hyperthreading), 16 GB RAM, 16x PCIe bus, a couple of Silicom dual port 10 Gigabit Ethernet NICs based on the Intel x540 chipset (8x PCIe), and Ubuntu 12.10 OS, kernel 3.5.0-17-generic, 64 bits. An entire CPU core is dedicated to the Master; instead, Workers have been allocated on the remaining CPU cores in a way that maximizes the throughput of the system. All the following graphs are obtained by averaging results of 100s tests repeated 10 times.

The data exchanged among the Master and the Workers consists of synthetic network packets of three sizes, 64 bytes to stress the forwarding capabilities of the chain, 700 bytes that matches the average packet size in current networks, and 1514 bytes to stress the data transfer capabilities of the system. We present first a set of experiments where packets exchanged between the Master and the Workers are directly read/written from/to the memory, without involving the network; those tests aim at validating the performance of the algorithm in isolation, without any disturbance such as the cost introduced by the driver used to access to the NIC or the overhead of the PCIe bus. Later, in the next sections, we will present some results involving a real network, where the workstation under test is connected with a second workstation acting as both traffic generator and receiver, with two 10Gbps dedicated NICs. This setup allows to derive the precise latency experienced by packets in our middlebox. In this case we use the PF_RING/DNA drivers [27] to read/write packets from/to the NIC, which allows the Master to send/receive packets without requiring the intervention of the operating system. In addition, data coming from the network is read in polling mode in order to limit additional overheads due to NIC interrupts, and in batches of several packets in order to maximize code locality. Similar techniques are used also when sending data to the network after all the processing took place.

Single chain - Throughput



(a) Dummy Workers and a single packet in memory.

(b) Real Workers and a single packet in memory.

(c) Dummy Workers and 1M packets in memory.

(d) Real Workers and 1M packets in memory.

Figure 2.5: Throughput of a single function chain with the algorithm presented in this section.

This section reports the performance of our algorithm in a scenario where all packets traverse the same chain, which is statically defined. Tests are repeated with chains of different lengths and the measured throughput is provided in graphs that include *(i)* a bars view corresponding to the left Y axis that reports the throughput in millions of packets per second and *(ii)* a point-based representation referring to the right Y axis, that reports the throughput in Gigabits per second.

Figure 2.5 shows the throughput offered by the function chain in different conditions. As expected, the overall throughput of the chain (i.e., the packets/bits that exit from the chain) decreases with the number of Workers because of our choice to reserve the most part of the CPU power to the Workers, hence limiting the Master to a single CPU core.

Figure 2.5(a) shows the throughput that could be achieved in ideal conditions, that is: *(i)* with dummy Workers, i.e., that do not touch the packet data, and *(ii)* with the Master always reading the same input packet from memory and copying it into the buffer of the first Worker of the chain, which reduces the overall number of CPU cache misses experienced at the beginning of the chain. This provides an ideal view of the system, where the penalties due to memory accesses are kept to a minimum. Results reported in Figure 2.5(b) are instead gathered in a more realistic scenario, i.e., with Workers that access to the packet content and calculate a simple signature across the first 64 bytes of packets. This may represent a realistic workload, as it emulates the fact that most network applications operate on the first bytes (i.e., the headers) of the packet. This test shows that performance is reduced compared to Figure 2.5(a) for two reasons: (*i*) because of the higher number of cache misses generated by the cores assigned to the Workers and caused by the Workers accessing to the packet content, and (*ii*) because of the additional processing time spent by the Workers for completing their job.

Next tests consider a scenario where the input data for the chain is stored in a buffer containing 1M packets, thus emulating a real middlebox that receives traffic from the network. In particular, Figure 2.5(c) refers to a scenario with dummy Workers such as in Figure 2.5(a) and shows how an apparently insignificant different memory access pattern can dramatically change the throughput. In fact, the Master experiences frequent cache misses when reading packets at the beginning of the chain. This modification alone halves the throughput compared to Figure 2.5(a), particularly when packets have to traverse chains of limited length, while in case of longer chains this additional overhead at the beginning is amortized by the cost of the rest of the chain.

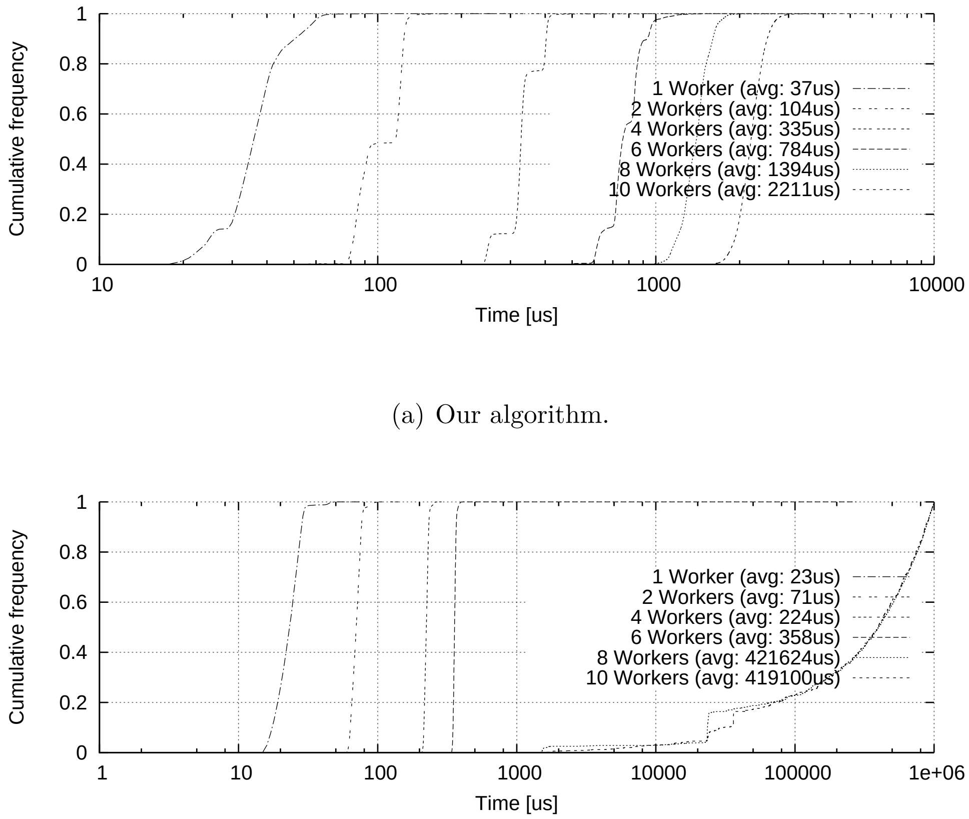
Finally, Figure 2.5(d) depicts a realistic scenario where Workers access the packet content (such as in Figure 2.5(b)), and the Master feeds the chain by reading data from a large initial buffer (1M packets). Even in this case our algorithm is able to guarantee an impressive throughput, such as about 38 Mpps with 64B packets.

In order to confirm that, with the current workload, the Master represents the bottleneck of the system, Figure 2.7 shows the internal throughput of the chain, namely the total number of packets moved by the Master, with an increasing number of Workers, in the same test conditions of Figure 2.5(d). This figure gives an insight of the processing capabilities of the Master, which slightly increases with a growing number of Workers and proves the effectiveness of our algorithm as the number of packets it processes essentially does not depend on the number of Workers.

2.1.10 Single chain - Latency

Some architectural and implementation choices, such as working with batches of packets, aim at improving the throughput but may badly affect the latency. For this reason, this section gives an insight about the latency experienced by packets traversing our chains. Measurements are based on the `gettimeofday` Unix system call and, in order to reduce its impact on the system, only sampled packets (one packet out of thousand) have been measured.

Figure 2.6(a) shows the latency of 64B packets when traversing a function chain consisting of a growing number of Workers, in case of real Workers and 1M packets in memory. As expected, the latency increases with the length of the chain; however its value is definitely reasonable for most of networking applications, reaching an average value of about 2.2ms in case of 10 cascading Workers, being far less with shorter (and more realistic) chains.



(a) Our algorithm.

(b) Zero-copy buffers among the Master and the (polling) Workers.

Figure 2.6: Latency introduced by the function chain with a growing number of cascading Workers.

2.1.11 Single chain - Comparison with other approaches

This section aims at demonstrating the advantages of our data exchange algorithm by comparing our proposal with two other approaches that could be used to exchange packets between the Master and the Workers.

In this respect, we cannot directly compare our algorithm with existing solutions such as VALE [71], OVS [64] and xDPd [6], because they include the overhead of packet classification (e.g., L2 forwarding, Openflow matching), which would affect the performance of the data exchange algorithm. As a consequence, we distilled the fundamental design choices of the most important alternative approaches and we carefully implemented them in a way that they could be compared with our algorithm, implemented by using, whenever applicable, the guidelines listed in Section 2.1.7. Particularly, the comparison aims at validating the advantages of two important aspects of our algorithm: the absence of a data copy in the Worker, and the blocking mode operating model of the Worker, while other approaches adopt a busy waiting model.

The first alternative approach we compare with is based on the traditional producer/consumer paradigm, in which the Master shares two buffers with each Worker: the first is used by the Master to provide packets to the Worker, while the second operates in the opposite direction. The second approach closely follows the processing model suggested by Intel in the DPDK library [36], namely, two buffers (based on the traditional producer/consumer paradigm) are shared between the Master and each Worker. However, these buffers contain pointers, which means that the actual data is stored in a shared mempool and never moved between the components of the function chain (zero-copy). Moreover, both the Master and Workers operate in polling mode. Although this solution neither provides isolation among the Workers, nor limits the CPU consumption, it is compared with our proposal because

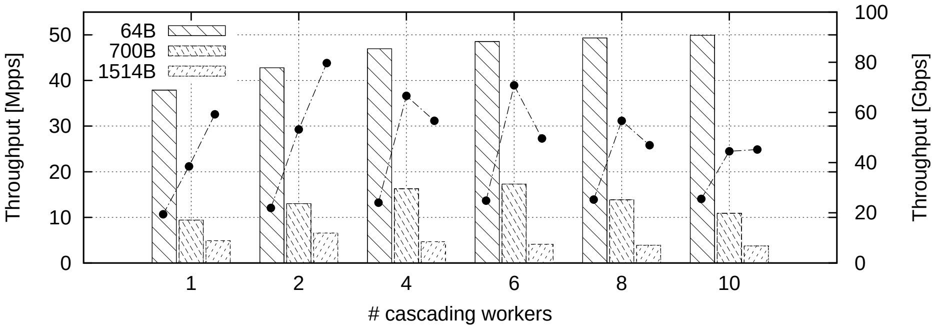
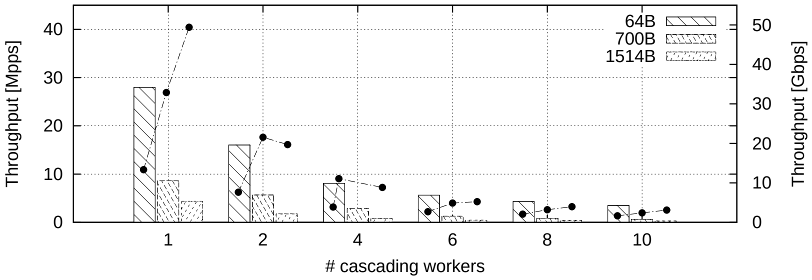
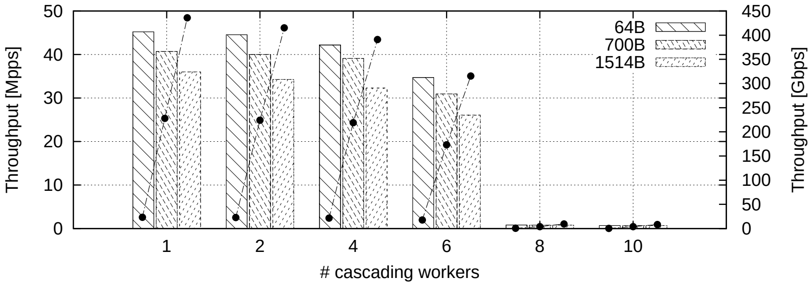


Figure 2.7: Internal throughput of the function chain, with real Workers and a 1M packets in memory.



(a) Unidirectional buffers shared between the Master and the Workers.



(b) Zero-copy buffers among the Master and the (polling) Workers.

Figure 2.8: Throughput of a single function chain when other data exchange algorithms are used.

nowadays it represents the "standard" way to implement network function chains.

Tests are executed in realistic conditions, namely with Workers accessing packets and 1M packets in memory and therefore the above results should be compared with the performance obtained in Figure 2.5(d).

As expected, the throughput of the chain drops of about 30% when unidirectional buffers are used, as shown by comparing Figure 2.5(d) and Figure 2.8(a). This is mainly due to the operating principles of our primary buffer, which allows the Worker to send back a packet to the Master without moving the packet itself, while in this alternative approach one additional data copy in the Worker has to be performed.

Instead, the second alternative approach slightly outperforms our algorithm until the number of jobs (one Master plus N Workers) is lower than the number of available CPU cores, as evident by comparing Figure 2.8(b) with Figure 2.5(d). This is due to the absence of data copies and to the polling-based operating mode implemented in the Workers. However, a stronger performance degradation with respect to our solution (it offers less than 1 Mpps throughput) is noticeable when 8 (or more) Workers are active because at least two of them have to share the same CPU core.

The second alternative approach has also been evaluated in terms of latency introduced on the flowing packets. Similarly to what happens for the throughput, it outperforms our proposal when the number of jobs running is less than the number of CPU cores, as evident by comparing Figure 2.6(a) and Figure 2.6(b). For instance, six chained Workers introduce an average latency of 358μs, against the 784μs obtained with our algorithm. Instead, in case of more Workers, the average latency of the second alternative approach reaches 420ms, which is a consequence of the fact that many polling processes share the same CPU core, and is definitely not acceptable. Hence, this solution neither provides isolation among Workers (due to the zero-copy), nor acceptable performance when the number of Workers exceeds the number of available cores, being inappropriate for our objectives.

2.2 The problem of checking SDN/NFV networks

Portions of this section were previously published in [81].

2.2.1 Introduction

Paradigms such as Software Defined Network (SDN) and Network Function Virtualization (NFV) can be seen as expressions of a systemic trend called “Softwarization". Other expressions of the same trend are Cloud, Edge, and Fog Computing, Cloud Networking, etc. In essence, the disruptive innovation of “Softwarization" stands in the techno-economic feasibility of virtualizing most (if not all) network and service functions of Telecommunications and ICT infrastructures. In this directions, it is argued that future Telecommunications infrastructures are likely to become highly dynamic, flexible and programmable production environments of ICT services. This flexibility, introduced by the SDN/NFV related technologies, has an important impact on the way services are actually developed and operated by network providers. While it is true that the network becomes an active production environments where multiples functionalities can be automatically provided by the infrastructure, i.e. without the operator intervening on all the steps needed to provide a new service, it is equally true that automatically deployed network configurations must be checked *before* they are installed in the production network so as to avoid configuration errors or malicious attempts to make the network unusable. A first evaluation of this idea is carried out by the EU FP7 UNIFY[3] consortium, which sets out to integrate modern cloud computing and networking technologies by considering the entire network as a unified service production environment, spanning the vast networking assets and data centers of telecom providers. In order to reach a high level of agility for service innovation, UNIFY has one focus on providing dynamic service programming and orchestration, deploying logical service components, namely Virtual Network Functions (VNFs), across multiple network nodes. In particular, UNIFY architecture follows SDN principles with a logically centralized control and orchestration plane. Additionally, compute, storage and network abstractions are combined into a joint programmatic interface referred to as Network Function Forwarding Graph (NF-FG). An NF-FG defines a selected mapping of VNFs and their forwarding overlay definition into the virtualized resources presented by the underlying layer. Current OSS/BSS do not seem to cope with the requirements posed by this evolution as they were basically designed for operating monolithic, closed physical nodes, involving significant deployment, integration and maintenance efforts: in fact, the operations of future Telecommunications infrastructures will involve the management and control of a myriad of software processes, rather than closed physical nodes. Thus, another important goal of UNIFY is the design and development of integrated operations and development capabilities under the name of Service Provider-DevOps (SP-DevOps). In fact, DevOps paradigm, formerly developed for Data Centers (DCs), is getting momentum as a source of inspiration regarding how to simplify and automate management processes for future Telecommunications infrastructures.

Among the above challenges, we focus on the UNIFY verification process (i.e., the definition of methods and techniques to validate a particular network configuration before deploying it), which can be seen as an essential task in environments where reconfiguration of services is expected to be triggered very frequently, both in response to user requests and also in case of management events. Misconfiguration of dynamic network middleboxes[4], violation of specified network policies, or artificial insertion of malicious network functions are just examples of cases that a complete solution must properly handle in order to preserve network integrity and reliability. For this reason, the work presented in this section goes in the direction of verifying complex graph of services through an intense modeling activity, targeted at the specific middleboxes and the network as a whole. We are motivated by the observation that most existing tools are “Openflow oriented", i.e. they mostly consider networks with a controller which installs <match, action> rules on the switches. Alternatively (and more generically but with the same fundamental limitations), they consider networks with devices that only perform forwarding decisions according to

[3] www.fp7-unify.eu

[4] In this section we use the terms VNF and middlebox interchangeably.

the packet header, i.e. without taking into account any additional traffic history information. Works as [43, 42, 66, 79] fall in this category and represent a valuable efforts in this research area. Our contribution is intended to move a step forward and overcome the above mentioned limitations by extending these works. In this sense, one important reference is [61], which tackles exactly the same problem and provides a scalable solution based on an off-the-shelf SMT solver. We experiment with this approach and further develop it to meet our specific requirements, also enriching the available VNF models catalog in order to satisfy the demands for more and more complex service graphs and to validate the approach with different kinds of VNFs. We specifically consider the UNIFY use cases, but it is worth noticing how our work is much general and easily applicable to other scenarios since it involves very common network functions.

2.2.2 The SP-DevOps concept

In order to cope with the high service velocity and increased dynamicity enabled by UNIFY and comparable SDN/NFV based environments, we consider a novel management and operation paradigm for Service Providers, called Service Provider DevOps - SP-DevOps. SP-DevOps is based on the same major underlying principles as identified for DevOps [75]:

- *i*) **Monitor** and validate operational quality;
- *ii*) **Develop** and test against production-like systems;
- *iii*) **Deploy** with repeatable, reliable processes;
- *iv*) **Amplify** feedback loops.

While we acknowledge that DevOps has also a crucial cultural dimension (reflected barely by the feedback loop principle), our work focuses on technical aspects associated to these principles, which reflect on processes and associated capabilities for integrated monitoring, verification, and testing software and programmable infrastructure. Even if significant parts of the telecommunication networks are foreseen to be virtualized in the future, we in [40] identified important characteristics of telecommunication networks that differ from traditional data centers, i.e.: (*i*) higher spatial distribution, as telecom resources are spread over wide areas due to coverage requirements; (*ii*) lower levels of redundancy in access and aggregation networks compared to the massive data centers of typical cloud computing companies; (*iii*) stronger requirements on high availability and latency in according to standards and customer expectations. These characteristics pose new challenges for applying DevOps principles in telecommunications environments [41]. SP-DevOps addresses them with a set of technical processes supporting developer and operator roles in a virtualized

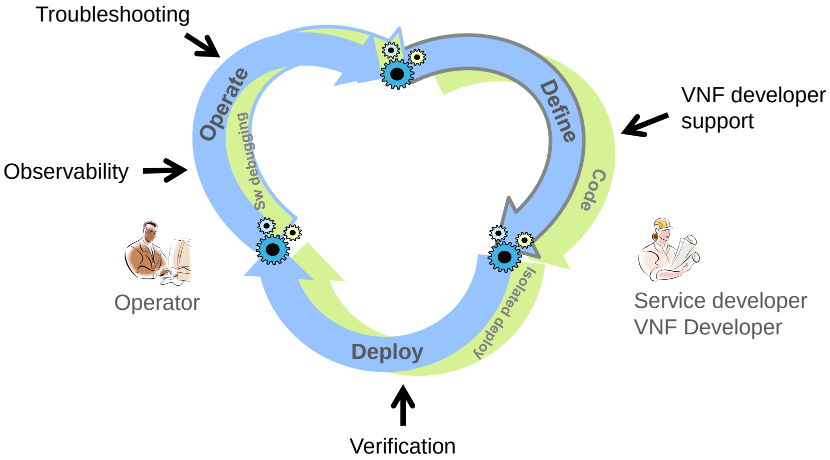


Figure 2.9: SP-DevOps cycle for UNIFY service creation.

telecom network. Figure 2.9 illustrates the relation between the four SP-DevOps processes and the developer/operator roles by means of a service creation lifecycle. The four SP-DevOps processes follow the DevOps principles to meet specific challenges regarding Observability and Troubleshooting (Principle: Monitor and validate operation quality); Verification (Principle: Deploy with repeatable, reliable processes); and Development (Principle: Develop and test against production-like systems). We also identified three main roles involved in the processes: two Developer roles, where one is associated to a classical operator role assembling the service graph for a particular category of services (the Service Developer), and a second associated to the classical equipment vendor role in actually programming a VNF (the VNF Developer). The role of the Operator is to ensure that a set of performance indicators associated to a service are met when the service is deployed on virtual infrastructure within the domain of a telecom provider. SP-DevOps might not be a new form of DevOps as such, but it must include solutions that are uniquely tailored for the characteristics of its environment. Consequently, we propose the SP-DevOps Toolkit as an instantiation of the SP-DevOps concept [56]. The SP-DevOps Toolkit consists of a set of DevOps solutions that are developed targeting specific research challenges identified in the UNIFY production environment [41, 40]. Besides scalable and programmable infrastructure monitoring functions, the toolkit will also provide modules for deploy-time functional verification of various abstraction levels of service definition, supporting the three SP-DevOps roles. As in any development process, identification of problems early in the service or product livecycle can significantly reduce times and costs spent on complicated debugging and troubleshooting processes. In the next sections, we focus on verification with respect to the service definitions and configurations initiated by the Service Developer. Automated verification functions operating during deploy-time on each layer of the orchestration and control architecture, facilitate verification as part of each step in the deployment process, allowing identification of problems early in the service lifecycle.

2.2.3 The verification process

The SP-DevOps paradigm represents a significant opportunity for service providers to implement more complex services in their networks and increase the agility by which a new function (or a chain of) can be automatically configured and deployed in their infrastructure. However, while the process of inserting and/or modifying functions throughout the network can be automated with technologies similar to the ones used for the Cloud Computing scenario [39], great importance has also to be placed on the design and implementation of automatic tools that can verify a network configuration on the fly, *before* it is deployed. This guarantees to avoid unexpected behaviours or that some desired properties are no longer satisfied in the new configuration. For example, an operator may want to ensure that a given traffic flow is permitted (or not permitted, due to a policy constraint) from one node to another. Concerning this last aspect, our verification process is currently based on a verification approach recently proposed in [61]. In order to achieve high performance, this verification approach exploits Z3 [18], a state of the art SMT solver, and translates network scenarios with multiple middleboxes into sets of First Order Logic (FOL) formulas that are then analyzed by Z3. This choice is motivated by the overall verification tool performance and scalability, which would be hard to achieve with standard model checking based techniques. In fact, the latter requires time and memory that usually increase exponentially with the system complexity, while the SAT-based approach proposed in [61] seems to be less prone to this problem. In the SP-DevOps context, where development cycles are rather fast, being able to analyze the new network configurations quickly is crucial, and this is the reason why we think the approach presented in [61] is promising for this kind of application. The FOL formulas given to Z3 represent the network operating principles along with the functional behavior of all the VNFs involved in the scenario being considered. While [61] presents the general ideas of the proposed approach, not all the details are fully developed, and not all the different situations that may arise when considering different kinds of VNFs are considered. Here, we present our preliminary work towards integrating the approach presented in [61] into a SP-DevOps context like the one of UNIFY. A considerable part of this work has been about developing models for new VNFs that were not explicitly considered in [61], and making some first experiments with them.

In our design, the formal verification task is split into multiple sub-tasks, so that the whole process is simpler and faster. More precisely, at NF-FG deploy time, or when the graphs undergo modifications in response to higher level events (e.g., administration events or user requests), the VNF chains composing the graph are computed and then, for each of them, a formal model is generated, including the model of all the involved VNFs. Finally, the verification engine processes the whole VNF chain model to check the satisfiability of a given property. In particular, we focus on reachability problems in service graphs, leaving the verification of other network properties as possible future work. Furthermore, since we are using abstract models of the real middleboxes, we assume that these models are correctly defined. This means that we verify abstract models of the real middleboxes, considering them as faithful representations of the real VNFs. Verification of possible mismatch between a VNF model and its implementation is out of scope for the current prototype. For further details about the adopted formal verification theory and other background concepts, please refer to [61].

VNFs models

The approach for modeling network function chains proposed in [61] has been experimented by the authors with some middlebox types, such as stateless and stateful firewalls. When modelling scenarios that include VNFs that may alter packets (e.g. a NAT), the network constraints specifications need to be changed, because, in the context of a reachability verification process, it is necessary to also consider the possibility for a target VNF to receive a packet different from the one originally transmitted. This kind of situation, which was not addressed explicitly in [61], regards a significant set of middleboxes that is currently deployed in SP networks and that is envisioned to be included in the NF-FG within the UNIFY project, e.g. NAT, VPN gateway and so on. We revisited the network constraints developed by the authors of [61], by introducing the possibility of verifying reachability properties between two network nodes and intermediate VNFs that do modify forwarded packet headers. Finally, we checked that verification works as expected with these revisited constraints, by experimenting with the new middlebox models that we developed.

The first one we consider is a simplified version of an anti-spam NF, whose model is reported in Figure 2.10.

We simplify the email protocol by assuming each client interested in receiving a new message addressed to him, sends a `POP3_REQUEST` to the mail server in order to retrieve the message content. The server, in turn, replies with a `POP3_RESPONSE` which contains a special field (named *email_from*) representing the message sender. The process of sending an email is similarly modeled through SMTP request and response messages. In order to be able to introduce our model within the verification tool, we need to express our function behaviour by means of logic constraints and implications. As evident from Formula 2.1a, our function rejects any message containing a black listed email address. On the other hand, Formula 2.1b is needed in order to state that a `POP3_REQUEST` message is forwarded only after having received

$$\begin{aligned}(send(antispam, n_0, p_0, t_0) \land p_0.protocol = POP3\_RESPONSE) \implies \\ \neg isInBlackList(p_0.email\_from) \land \exists(n_1, t_1) \mid (t_1 < t_0 \\ \land\, recv(n_1, antispam, p_0, t_1)) \\ \forall n_0, p_0, t_0\end{aligned} \tag{2.1a}$$

$$\begin{aligned}(send(antispam, n_0, p_0, t_0) \land p_0.protocol = POP3\_REQUEST) \implies \\ \exists(n_1, t_1) \mid (t_1 < t_0 \\ \land\, recv(n_1, antispam, p_0, t_1)) \\ \forall n_0, p_0, t_0\end{aligned} \tag{2.1b}$$

Figure 2.10: Antispam model

$$\begin{aligned}&(send(cache, n_0, p_0, t_0) \land \neg isInternal(n_0)) \implies \neg isInCache(p_0.url, t_0) \\ &\quad \land p_0.proto = HTTP\_REQ \land \exists(t_1, n_1) \mid (t_1 < t_0 \land isInternalNode(n_1) \\ &\quad \land recv(n_1, cache, p_0, t_1)), \forall n_0, p_0, t_0\end{aligned} \tag{2.2a}$$

$$\begin{aligned}&(send(cache, n_0, p_0, t_0) \land isInternal(n_0)) \implies isInCache(p_0.url, t_0) \\ &\quad \land p_0.proto = HTTP\_RESP \land p_0.ip\_src = p_1.ip\_dest \land p_0.ip\_dest = p_1.ip\_src \land \\ &\quad \land \exists(p_1, t_1) \mid (t_1 < t_0 \land p_1.protocol = HTTP\_REQ \land p_1.url = p_0.url \\ &\quad \land recv(n_0, cache, p_1, t_1)), \forall n_0, p_0, t_0\end{aligned} \tag{2.2b}$$

$$\begin{aligned}&isInCache(u_0, t_0) \implies \exists(t_1, t_2, p_1, p_2, n_1, n_1) \mid (t_1 < t_2 \land t_1 < t_0 \land t_2 < t_0 \\ &\quad \land recv(n_1, cache, p_1, t_1) \land recv(n_2, cache, p_2, t_2) \land p_1.proto = HTTP\_REQ \\ &\quad \land p_1.url = u_0 \land p_2.proto = HTTP\_RESP \land p_2.url = u_0 \land isInternal(n_2)) \\ &\quad \forall u_0, t_0\end{aligned} \tag{2.2c}$$

Figure 2.11: Web cache model.

it in a previous time instant. As it is clear from this description, the anti-spam function is completely stateless: it only needs information contained within the scope of a single packet to decide how to process it (i.e. forward or drop). As next step, we introduce two more models that are heavily based on traffic history, namely VNFs whose behaviour may be altered by a proper sequence of packets.

Another VNF we consider is a simple web cache (reported in Figure 2.11). The functional model consists of two interfaces connected respectively to the private network, i.e., the one which contains the clients issuing HTTP requests, and the external network.

Formula 2.2a states that a packet sent from the cache to a node belonging to the external network, implies a previous packet, containing a HTTP request and received from an internal node, which cannot be served by the cache (otherwise the request would have not been forwarded towards the external network). Formula 2.2b states that a packet sent from the cache to the internal network contains a `HTTP RESPONSE` for an URL which was in cache when the request has been received. We also state that the packet received from the internal network is a `HTTP REQUEST`

$$\begin{aligned}
&(send(nat, n_0, p_0, t_0) \land \neg isPrivateAddress(p_0.ip\_dest)) \implies p_0.ip\_src = ip\_nat \\
&\quad \land \exists (n_1, p_1, t_1) \mid (t_1 < t_0 \land recv(n_1, nat, p_1, t_1) \land isPrivateAddress(p_1.ip\_src) \\
&\quad \land p_1.origin = p_0.origin \land p_1.ip\_dest = p_0.ip\_dest \land p_1.seq\_no = p_0.seq\_no \\
&\quad \land p_1.proto = p_0.proto \land p_1.email\_from = p_0.email\_from \land p_1.url = p_0.url) \\
&\quad \forall n_0, p_0, t_0
\end{aligned} \tag{2.3a}$$

$$\begin{aligned}
&(send(nat, n_0, p_0, t_0) \land isPrivateAddress(p_0.ip_d est)) \implies \neg isPrivateAddress(p_0.ip\_src) \\
&\quad \land \exists (n_1, p_1, t_1) \mid (t_1 < t_0 \land recv(n_1, nat, p_1, t_1) \land \neg isPrivateAddress(p_1.ip\_src) \\
&\quad \land p_1.ip\_dest = ip\_nat \land p_1.ip\_src = p_0.ip\_src \land p_1.origin = p_0.origin \\
&\quad \land p_1.seq\_no = p_0.seq\_no \land p_1.proto = p_0.proto \land p_1.email\_from = p_0.email\_from \\
&\quad \land p_1.url = p_0.url) \land \exists (n_2, p_2, t_2) \mid (t_2 < t_1 \land recv(n_2, nat, p_2, t_2) \\
&\quad \land isPrivateAddress(p_2.ip\_src) \land p_2.ip\_dest = p_1.ip\_src \land p_2.ip\_dest = p_0.ip\_src \\
&\quad \land p_2.ip\_src = p_0.ip\_dest), \forall n_0, p_0, t_0
\end{aligned} \tag{2.3b}$$

Figure 2.12: NAT model.

and the target URL is the same as the response. The final formula expresses a constraint that the *isInCache()* function must respect. In particular, we state that a given URL (u_0) is in cache at time t_0 if (and only if) a request packet was received at time t_1 (where $t_1 < t_0$) for that URL and a subsequent packet was received at time t_2 (where $t_2 < t_0 \land t_2 > t_1$) carrying the corresponding `HTTP RESPONSE`.

It is worth noticing that this model is a simplified version of a real web cache since, if our function receives a request for a particular URL and that content is already in cache, no additional packets are generated in order to be sure that no update exists for that content (as usually done by means of the If-Modified-Since HTTP header) and the response is immediately forwarded. We neglect these details in order to keep the cache model as much clear and straightforward as possible. The last middlebox we present here is the NAT function. The corresponding model is reported in Figure 2.12.

In order to model the NAT behaviour, a distinction between the private and external network is needed. This separation is modeled by using a boolean function (*isPrivateAddress()*) that returns true if a given IP address belongs to the set of internal node addresses. Analyzing the reported formulas, we start by considering an internal node which initiates a communication with an external node (Formula 2.3a). In this case, the NAT sends a packet (p_0) to an external IP address, if and only if it has previously received a packet (p_1) from an internal node. The received and sent packets must be equal for all fields, except for the *ip_src*, which must be equal to the NAT public IP address.
On the other hand, the traffic in the opposite direction (from the external network to the private) is modeled by the Formula 2.3b. In this case, we state that if the NAT is sending a packet to an internal address, this packet (p_0) must have an external IP address as its source. Moreover, p_0 must be preceded by another packet (p_1 in

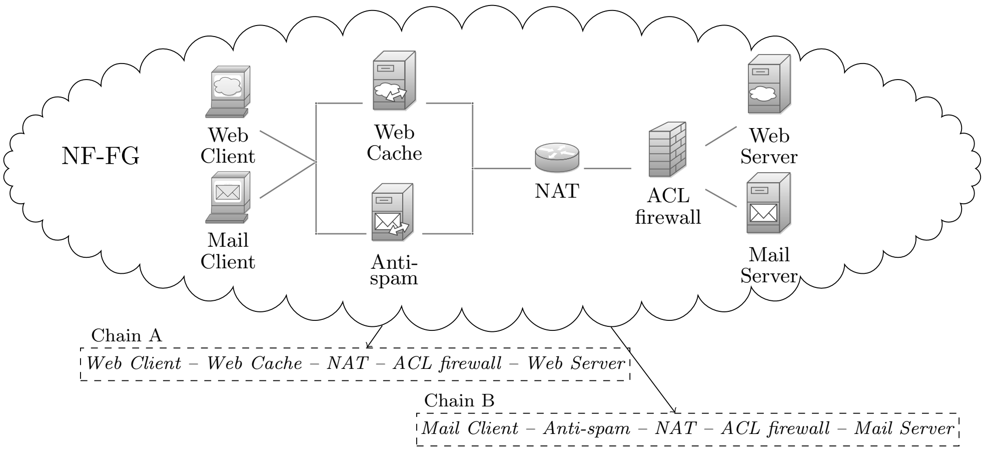

NF-FG
Web Client
Mail Client
Web Cache
Anti-spam
NAT
ACL firewall
Web Server
Mail Server
Chain A
Web Client – Web Cache – NAT – ACL firewall – Web Server
Chain B
Mail Client – Anti-spam – NAT – ACL firewall – Mail Server

Figure 2.13: An example of Network Function-Forwarding Graph.

the formula), which is, in turn, received by the NAT and it is equal to p_0 for all the other fields. It is worth noting that, generally, a communication between internal and external nodes cannot be started by the external node in presence of a NAT. As a consequence, this condition is expressed in Formula 2.3b by imposing that p_1 must be preceded by another packet p_2, sent to the NAT from an internal node.

2.2.4 Verification results

In order to evaluate the new developed models and the overall approach, we consider the NF-FG[5] shown in Figure 2.13 as a use case. In our reference graph, four end-hosts (two clients and two servers) can generate either HTTP or POP3 and also SMTP traffic, which is processed by different middleboxes when traversing the graph. Moreover, some of those network functions may require a different configuration. Specifically, the NAT must be configured in order to know which hosts belong to the private network (as the web cache) and which IP address must be used as masquerading address; the firewall must be provided with a set of ACL entries that specify which couples of nodes are authorized to exchange traffic. Additionally, the forwarding is configured such that the web traffic is forwarded to the web cache, while the email traffic (both POP3 and SMTP) is routed to an anti-spam function.
A first step towards the NF-FG verification is the VNF chains extraction. In our use case, two chains are extracted from the NF-FG (Figure 2.13): the *Chain A* processes the web traffic, while the *Chain B* is traversed by POP3 and SMTP packets.
We perform multiple tests on the two chains to cover different cases and configuration options: *(i)* anti-spam and firewall configurations and *(ii)* traffic directions (from

[5] We do not provide the firewall VNF model as it was presented as use case in [61].

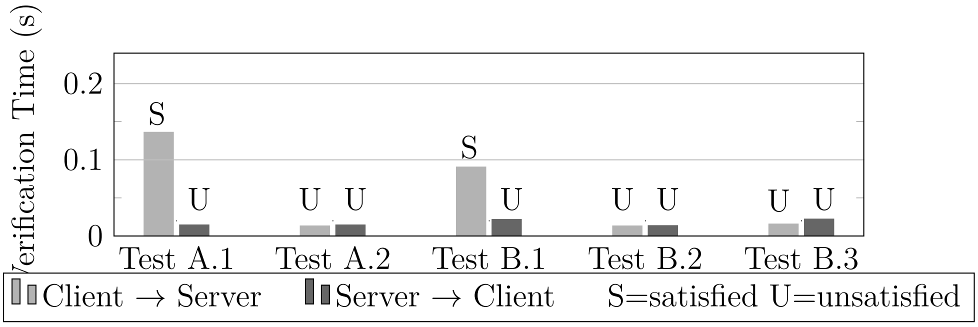


Figure 2.14: **Test {A, B}.1**: firewall and anti-spam configured to accept packets; **Test {A, B}.2**: firewall configured to drop server/client packets; **Test B.3**: anti-spam configured to drop server/client packets.

client to server and vice-versa). Concerning the Chain A, only the ACL firewall can be configured, hence we setup two tests: one with the firewall configured to allow all the traffic (test A.1) and the other one with the firewall configured to drop all packets exchanged between the web client and server (test A.2).
Instead the Chain B is tested in three scenarios, obtained by changing the firewall and anti-spam configurations as follows:

- (*i*) test B.1, similarly to test A.1, is performed without any function configured to drop the received traffic;
- (*ii*) in test B.2, the firewall drops the traffic between the mail client and server (Figure 2.14);
- (*iii*) test B.3 is such that the anti-spam is configured to drop all the emails sent by the mail client, while the traffic originated by the server is allowed (Figure 2.14).

Our evaluation is executed on a workstation with 32GB of RAM and an Intel i7-3770 CPU running an Ubuntu 14.04.01 with kernel 3.13.0-24-generic. The results are shown in Figure 2.14, where the verification time is reported for each presented scenario.
In test A.1 the reachability problem from the client to the server (the light grey colored bar in Figure 2.14) is satisfied as expected. It is worth noting that the unsatisfiability of the problem in the opposite direction (the dark grey colored bar in Figure 2.14) is due to the fact that client and server can exchange traffic only if the connection is initiated by the client. In test A.2, in both cases the reachability problems are not satisfied because of the firewall VNF configuration. In test B.1, the verification problem is satisfiable in case of traffic sent by the mail client, while the reachability property is not verified for the traffic sent by the mail server for the

above-mentioned reasons.
As it can be seen from the achieved results, performance is promising also in the worst case scenario, since we are able to solve the reachability problem in less than 200ms, while the verification time is less than 50ms in most cases. This is reasonably in line with the UNIFY requirements, especially in terms of time required by the verification process to authorize a newly asked network reconfiguration.

2.2.5 Discussion

It is argued that, in the future, Telecommunications infrastructures are likely to become highly dynamic, flexible and programmable production environments capable of providing any ICT services. Current OSS/BSS do not seem to cope with the requirements posed by this evolution: in fact, future operations will involve the management and control of a myriad of software processes, rather than closed physical nodes.

In fact, today most SPs still have rather complicated and static operational environments for legacy infrastructures, where IT and Network are fully separated, and managed by different Department. DevOps, formerly developed for managing Data Centers (DCs), is attracting a growing interests as a paradigm to be extended to future Telecommunications infrastructures. Nevertheless, it is argued that the DevOps will jump ahead current ossification only if it will be sustainable from a business viewpoint (CAPEX, OPEX saving are not enough): importantly DevOps criteria of success depend on how closely the related future infrastructures (e.g. UNIFY) will be capable of enabling new service paradigms for SP's (e.g., Immersive Communications, Anything as a Service, etc) and to create new ICT ecosystems (e.g., coming from the crossing of Telecommunications with Internet of Thing, or Cloud Robotics). Motivated by these considerations, we presented our contribution related to the verification process on service graphs, which is one of the most important pillars in the SP-DevOps feedback cycle and a key enabler to support envisioned changes in the way SPs deploy and operate new network services. After generalizing the applicability of a state of the art approach to the verification of complex network graph, we presented and discussed different models we developed to validate our key ideas.

Chapter 3

Conclusion

The evolution of the Internet is posing serious challenges to the scientific community from different sides: content dissemination, network services flexibility, network programmability, softwarization and data-plane performance scalability issues. The success of these new paradigms, that we extensively analyzed in this work, is strictly related to their ability to provide value-added services to end customers, i.e. tangible benefits in terms of user experience and/or in terms of rich features set. On the other hand, most of the newly proposed network architectures will be key in the next future only if the network usage performed by end users will evolve accordingly. Indeed our understanding is that any protocol or architecture for the future Internet should focus not only on currently available trends and services but has to be general enough to support *future* possibilities and scenarios. As a lesson learned from the past, one of the key features of the TCP/IP protocol suite is to be general enough to support any data exchange the community could think of at that time and also low demands for the lower, physical layers which is clue to split the complexity on the proper layers. These design choices made most of the IP related protocols, the first class citizens in the modern Internet and they are still in place after decades of fast changing technologies.

Among the new protocols and architectures that have been recently proposed, one important aspect that must be carefully assessed is the security and reliability of all of them. As we have shown in this thesis, differentiated weaknesses must be closely evaluated and quantified by means of different techniques: a simulation-based approach can be seen as a convenient way to represent the most common operating scenarios and it often provides an effective technique to catch the essence of a given system/protocol, that is the most important benefits (defined and measured in some way) and also the possible drawbacks. On the other hand, more formal techniques can be used to achieve rigorous results with a solid mathematical foundation. In this work, we exploited either techniques to highlight problems and the corresponding solutions, also trying to detect the most promising architectures for the future Internet given the ever evolving environment of modern telecommunication networks.

Regarding CCN, we analyzed and quantified a new family of DDoS attacks that can be implemented on a content-oriented network. After evaluating this important issue in realistic network conditions, we also selected the state of the art countermeasures and implemented them in our simulation tool in order to detect the most effective technique at Internet scale. Concerning scalability, we have found huge challenges in the implementation of PUSH based applications in CCN environments. Motivated by the need of enabling support to this kind of applications also in content-oriented networks, we proposed an architecture to overcome existing limitations especially in terms of protocol routing scalability and we accurately evaluated the additional overhead needed to support content delivery. We believe this is a key feature since a wide part of modern applications require producer driven communications.

Innovation in current telecommunication networks is inspired not only by an ever increasing demand for efficient content distribution but also by the new opportunities created by virtualization technologies. In other words, the focus will be more and more moved to the *transformation*[1] of bits during their trip along the network and not only on their *transportation*. Perfectly aligned with this vision, we studied innovative architectures to implement the Service Function Chaining paradigm. Our contribution took the form of an algorithm to efficiently move packets in NFV/SDN contexts, namely a mechanism to allow fast packets processing among different VNFs. Concerning VNFs, we also proposed an enhanced general framework (based on state of the art solutions) to rigorously verify critical network policies in order to avoid inconsistent configurations that could break network coherency. Our tool is a full fledged verification module, designed to be included in a wider set of components that are together able to automate the process of developing and operating new services in telecom operator networks.

Progresses in telecommunication networks and wider and wider availability of standard, open source solutions are leading remarkable changes in the ICT ecosystem. As a matter of fact, protocols, architectures and software solutions must keep pace to support society in this never ending effort to improve the quality that ICT can safely bring in our everyday life. In the next future, our ability to develop smart solutions will define the boundaries of the collective progress that we, as society, will pursue.

[1] *Here this term means offering services to end users or enterprises in a customizable way with the explicit support of the network operator.*

Bibliography

[1] A. Afanasyev, P. Mahadevan, I. Moiseenko, E. Uzun, and Lixia Zhang. Interest flooding attack and countermeasures in named data networking. In *IFIP Networking Conference, 2013*, pages 1–9, May 2013.

[2] A. Afanasyev, I. Moiseenko, and L. Zhang. ndnsim: Ndn simulator for ns-3. *Technical Report, NDN-0005*, Oct. 2012.

[3] S. Arianfar, P. Nikander, and J. Ott. On content-centric router design and implications. In *ReARCH '10*, Philadelphia, PA, USA, Nov. 2010.

[4] E. Baccaglini, G. Marchetto, T. Tillo, and G. Olmo. Efficient slice-aware h.264/avc video transmission over time-driven priority networks. *International Journal of Communication Systems*, 27(12):3822–3836, 2014.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[6] BISDN. xdpd. `http://www.xdpd.org`, 2016.

[7] Jeremias Blendin, Julius Rückert, Nicolai Leymann, Georg Schyguda, and David Hausheer. Position paper: Software-defined network service chaining. In *Proceedings of the Third European Workshop on Software Defined Networking (EWSDN 2014)*, 2014.

[8] Antonio Carzaniga, Koorosh Khazaei, Michele Papalini, and Alexander L. Wolf. Is information-centric multi-tree routing feasible? In *Proceedings of the 3rd ACM SIGCOMM Workshop on Information-centric Networking*, ICN '13, pages 3–8, New York, NY, USA, 2013. ACM.

[9] I. Cerrato, G. Marchetto, F. Risso, R. Sisto, and M. Virgilio. An efficient data exchange algorithm for chained network functions. In *2014 IEEE 15th International Conference on High Performance Switching and Routing (HPSR)*, pages 98–105, July 2014.

[10] I. Cerrato, G. Marchetto, F. Risso, R. Sisto, and M. Virgilio. An efficient data exchange algorithm for chained network functions. In *High Performance Switching and Routing (HPSR), 2014 IEEE 15th International Conference on*, pages 98–105, July 2014.

[11] Ivano Cerrato, Tobias Jungel, Alex Palesandro, Fulvio Risso, Marc Suñé, and

Hagen Woesner. User-specific network service functions in an sdn-enabled network node. In *Proceedings of the Third European Workshop on Software Defined Networking (EWSDN 2014)*, pages 135–136, 2014.

[12] Jia Chen, Hongke Zhang, Huachun Zhou, and Hongbin Luo. Optimizing content routers deployment in large-scale information centric core-edge separation internet. *International Journal of Communication Systems*, 27(5):794–810, 2014.

[13] Jiachen Chen, M. Arumaithurai, Lei Jiao, Xiaoming Fu, and K.K. 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, Oct 2011.

[14] Seungoh Choi, Kwangsoo Kim, Seongmin Kim, and Byeong hee Roh. Threat of dos by interest flooding attack in content-centric networking. In *Information Networking (ICOIN), 2013 International Conference on*, pages 315–319, Jan 2013.

[15] Alberto Compagno, Mauro Conti, Paolo Gasti, and Gene Tsudik. Poseidon: Mitigating interest flooding ddos attacks in named data networking. *CoRR*, abs/1303.4823, 2013.

[16] Huichen Dai, Bin Liu, Yan Chen, and Yi Wang. On pending interest table in named data networking. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 211–222, New York, NY, USA, 2012. ACM.

[17] Huichen Dai, Yi Wang, Jindou Fan, and Bin Liu. Mitigate ddos attacks in ndn by interest traceback. In *NOMEN '13*, Turin, Italy, Apr. 2013.

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Vladimir Dimitrov and Ventzislav Koptchev. Psirp project – publish-subscribe internet routing paradigm: New ideas for future internet. In *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, CompSysTech '10, pages 167–171, New York, NY, USA, 2010. ACM.

[20] Kun Ding, Yun Liu, Hsin-Hung Cho, Han-Chieh Chao, and Timothy K. Shih. Cooperative detection and protection for interest flooding attacks in named data networking. *International Journal of Communication Systems*, 2014.

[21] Constantinos Dovrolis, Brad Thayer, and Parameswaran Ramanathan. Hip: Hybrid interrupt-polling for the network interface. *ACM Operating Systems Reviews*, 35:50–60, 2001.

[22] D.Rossi and G. Rossini. Caching performance of content centric networks under

multi-path routing (and more). *Technical Report, Telecom Paris Tech*, 2011.

[23] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[24] European Telecommunications Standards Institute. Network Functions Virtualisation. White paper, SDN and OpenFlow World Congress, Darmstadt, Germany, Oct. 2012.

[25] Bohao Feng, Huachun Zhou, Shuai Gao, and Ilsun You. An exploration of cache collaboration in information-centric network. *International Journal of Communication Systems*, 2014.

[26] Nikos Fotiou, Dirk Trossen, and GeorgeC. Polyzos. Illustrating a publish-subscribe internet architecture. *Telecommunication Systems*, 51(4):233–245, 2012.

[27] Francesco Fusco and Luca Deri. High speed network traffic analysis with commodity multi-core systems. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 218–224, New York, NY, USA, 2010. ACM.

[28] P. Gasti, G. Tsudik, E. Uzun, and Lixia Zhang. Dos and ddos in named data networking. In *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*, pages 1–7, July 2013.

[29] Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In *Proceedings of the 14th international conference on Principles of distributed systems*, OPODIS'10, pages 302–317, Berlin, Heidelberg, 2010. Springer-Verlag.

[30] Moneeb Gohar, Heeyoung Jung, and Seok-Joo Koh. Distributed mapping management of identifiers and locators in mobile-oriented internet environment. *International Journal of Communication Systems*, 27(1):95–115, 2014.

[31] S. Govindan, Jeonghwan Choi, A.R. Nath, A. Das, B. Urgaonkar, and Anand Sivasubramaniam. Xen and co.: Communication-aware cpu management in consolidated xen-based hosting platforms. *Computers, IEEE Transactions on*, 58(8):1111–1125, Aug 2009.

[32] Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In Eduardo Tovar, Philippas Tsigas, and Hacene Fouchal, editors, *OPODIS*, volume 4878 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2007.

[33] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual.* Addison-Wesley Professional, first edition, 2003.

[34] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, May 1997.

[35] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation*

(NSDI 14), pages 445–458, Seattle, WA, 2014. USENIX Association.

[36] Intel. Data plane developer kit - programmers guide. http://www.dpdk.org, 2012.

[37] V. Jacobson, D. K. Smetters, N. H. Briggs, M. F. Plass, P. Stewart, J. D. Thornton, and R. L. Braynard. Networking named content. In *CoNEXT '09*, Rome, Italy, Dec. 2009.

[38] Van Jacobson, Diana K. Smetters, Nicholas H. Briggs, Michael F. Plass, Paul Stewart, James D. Thornton, and Rebecca L. Braynard. Voccn: Voice-over content-centric networks. In *Proceedings of the 2009 Workshop on Re-architecting the Internet*, ReArch '09, pages 1–6, New York, NY, USA, 2009. ACM.

[39] R. Jain and S. Paul. Network virtualization and software defined networking for cloud computing: a survey. *Communications Magazine, IEEE*, 51(11), November 2013.

[40] W. John and C. Meirosu. Unify d4.1: Initial requirements for the sp-devops concept, universal node capabilities and proposed tools, 2014.

[41] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu. Research directions in network service chaining. In *SDN4FNS, 2013 IEEE SDN for*, Nov 2013.

[42] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI 12*, San Jose, CA, 2012. USENIX.

[43] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI 13*, Lombard, IL, 2013. USENIX.

[44] Do-hyung Kim, Jong-hwan Kim, Yu-sung Kim, Hyun-soo Yoon, and Ikjun Yeom. End-to-end mobility support in content centric networks. *International Journal of Communication Systems*, 2014.

[45] Patrick P. C. Lee, Tian Bu, and Girish P. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *IPDPS*, pages 1–12, 2010.

[46] Lei Li, Xin Lin, Yue Zhai, Caixia Yuan, Yanquan Zhou, and Jiayin Qi. User communities and contents co-ranking for user-generated content quality evaluation in social networks. *International Journal of Communication Systems*, pages n/a–n/a, 2014.

[47] Pei Li and Yunchuan Sun. Modeling and performance analysis of information diffusion under information overload in facebook-like social networks. *International Journal of Communication Systems*, 2014.

[48] Jianxin Liao, Qi Qi, Tonghong Li, Yufei Cao, Xiaomin Zhu, and Jingyu Wang. An optimized qos scheme for ims-nemo in heterogeneous networks. *International Journal of Communication Systems*, 25(2):185–204, 2012.

[49] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins. Known issues and best

practices for the use of long polling and streaming in bidirectional http. *RFC 6202*, Apr. 2011.

[50] Joao Martins, Mohamed Ahmed, Costin Raiciu, and Felipe Huici. Enabling fast, network processing with clickos. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 67–72, New York, NY, USA, 2013. ACM.

[51] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, 2014. USENIX Association.

[52] H. Massalin and C. Pu. Threads and input/output in the synthesis kernal. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP '89, pages 191–201, New York, NY, USA, 1989. ACM.

[53] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. *SIGOPS Oper. Syst. Rev.*, 26(2):108–, April 1992.

[54] B. Mathieu, P. Truong, Wei You, and J. Peltier. Information-centric networking: a natural design for social network applications. *Communications Magazine, IEEE*, 50(7):44–51, July 2012.

[55] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[56] C. Meirosu. m4.1: Sp-devops concept evolution and initial plans for prototyping, 2014.

[57] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[58] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *USENIX Annual Technical Conference*, pages 99–112, 1996.

[59] NDN. Named Data Networking: A future Internet architecture. http://named-data.net/, 2014. [Online; accessed 13-May-2014].

[60] Shumao Ou, Hui Pan, and Feng Li. Heterogeneous wireless access technology and its impact on forming and maintaining friendship through mobile social networks. *International Journal of Communication Systems*, 25(10):1300–1312, 2012.

[61] Aurojit Panda, Ori Lahav, Katerina J. Argyraki, Mooly Sagiv, and Scott Shenker. Verifying isolation properties in the presence of middleboxes. *CoRR*, abs/1409.7687, 2014.

[62] PARC. Ccnx technical documentation. *http://www.ccnx.org/*, 2016.

[63] D. Perino and M. Varvello. A reality check for content centric networking. In *ICN '11*, Toronto, Canada, Aug. 2011.

[64] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. Extending networking into the virtualization layer. In *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, October 2009.

[65] P.Gasti, G. Tsudik, E. Uzun, and L. Zhang. Dos & ddos in named-data networking. *arXiv:1208.0952*, Aug. 2012.

[66] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, New York, NY, USA, 2012. ACM.

[67] S. Prakash, Yann Hang Lee, and T. Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Trans. Comput.*, 43(5):548–559, May 1994.

[68] Radware. Global application & network security report. *Annual Report*, 2011.

[69] Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner. Hyperswitch: A scalable software virtual switching architecture. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 13–24, Berkeley, CA, USA, 2013. USENIX Association.

[70] Fulvio Risso and Ivano Cerrato. Customizing data-plane processing in edge routers. In *Proceedings of the First European Workshop on Software Defined Networking (EWSDN)*, pages 114–120, 2012.

[71] Luigi Rizzo and Giuseppe Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM.

[72] O. Saleh and M. Hefeeda. Modeling and caching of peer-to-peer traffic. In *ICNP '06*, Santa Barbara, CA, USA, Nov. 2006.

[73] S.Arianfar, T. Koponen, B. Raghavan, and S. Shenker. On preserving privacy in content-oriented networks. In *ICN '11*, Toronto, Canada, Aug. 2011.

[74] Thomas C. Schmidt and Matthias Wählisch. Why we shouldn't forget multicast in name-oriented publish/subscribe. *CoRR*, abs/1201.0349, 2012.

[75] Sanjeev Sharma and Bernie Coyne. *DevOps For Dummies*. Limited IBM Edition' book, October 2013.

[76] Amanda Sibley. 47 handy Facebook stats and charts. http://boletines.prisadigital.com/47_facebook_stats_and_charts2.pdf, 2012. [Online; accessed 13-May-2014].

[77] We Are Social. Social, Digital & Mobile Around The World (January 2014). http://www.slideshare.net/wearesocialsg/

social-digital-mobile-around-the-world-january-2014, 2014. [Online; accessed 13-May-2014].

[78] A. Soldati. Telecom italia ip backbone and peering policies. In *Italian Peering Forum (PFI 2008)*, 2008.

[79] Sooel Son, Seungwon Shin, Vinod Yegneswaran, Phillip A. Porras, and Guofei Gu. Model checking invariant security properties in openflow. In *ICC*, pages 1974–1979. IEEE, 2013.

[80] Telecom Italia S.p.A. Resoconto intermedio di gestione al 31 marzo 2012 (in italian). *http://1q2012.telecomitalia.com/ attachments/telecomitalia2012q1it.pdf*, 2012.

[81] Serena Spinoso, Matteo Virgilio, Wolfgang John, Antonio Manzalini, Guido Marchetto, and Riccardo Sisto. *Service Oriented and Cloud Computing: 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015, Proceedings*, chapter Formal Verification of Virtual Network Function Graphs in an SP-DevOps Context, pages 253–262. Springer International Publishing, Cham, 2015.

[82] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '01, pages 134–143, New York, NY, USA, 2001. ACM.

[83] Christos Tsilopoulos and George Xylomenos. Supporting diverse traffic types in information centric networks. In *Proceedings of the ACM SIGCOMM Workshop on Information-centric Networking*, ICN '11, pages 13–18, New York, NY, USA, 2011. ACM.

[84] M. Virgilio, G. Marchetto, and R. Sisto. Interest flooding attack countermeasures assessment on content centric networking. In *Information Technology - New Generations (ITNG), 2015 12th International Conference on*, pages 721–724, April 2015.

[85] Matteo Virgilio. Shared buffer model. https://github.com/netgroup-polito/shared-buffer, 2015.

[86] Matteo Virgilio, Guido Marchetto, and Riccardo 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.

[87] Matteo Virgilio, Guido Marchetto, and Riccardo Sisto. Push applications and dynamic content generation over content-centric networking. *International Journal of Communication Systems*, pages n/a–n/a, 2015.

[88] Matthias Wählisch, Thomas C. Schmidt, and Markus Vahlenkamp. Backscatter from the data plane - threats to stability and security in information-centric network infrastructure. *Comput. Netw.*, 57(16):3192–3206, November 2013.

[89] Jin Wang, Kejie Lu, Shukui Zhang, Jianxi Fan, Yanqin Zhu, and Baolei

Cheng. An efficient communication relay placement algorithm for content-centric wireless mesh networks. *International Journal of Communication Systems*, 28(2):262–280, 2015.

[90] Kai Wang, Jia Chen, Huachun Zhou, Yajuan Qin, and Hongke Zhang. Modeling denial-of-service against pending interest table in named data networking. *International Journal of Communication Systems*, 2013.

[91] Qingjie Wang, Jianrong Wang, Jian Yu, Mei Yu, and Yan Zhang. Trust-aware query routing in p2p social networks. *International Journal of Communication Systems*, 25(10):1260–1280, 2012.

[92] Kun Yang, Xueqi Cheng, Liang Hu, and Jianming Zhang. Mobile social networks: state-of-the-art and a new vision. *International Journal of Communication Systems*, 25(10):1245–1259, 2012.

[93] W. You, B. Mathieu, P. Truong, J. Peltier, and G. Simon. Dipit: a distributed bloom-filter based pit table for ccn nodes. In *ICCCN '12*, Munich, Germany, July 2012.

[94] Haowei Yuan, Tian Song, and Patrick Crowley. Scalable ndn forwarding: Concepts, issues and principles. In *ICCCN '12*, Munich, Germany, July 2012.

[95] Li Zhao, L.N. Bhuyan, R. Iyer, S. Makineni, and D. Newell. Hardware support for accelerating data movement in server platform. *Computers, IEEE Transactions on*, 56(6):740–753, June 2007.