ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electronics and Telecommunications Engineering (30^{th}cycle)

Caching Techniques in Next Generation Cellular Networks

By

Ahsan Mahmood

******

Supervisor(s):
Prof. Carla-Fabiana Chiasserini, Supervisor
Prof. Paolo Giaccone, Co-Supervisor
Prof. Jerome Härri, Co-Supervisor

Doctoral Examination Committee:
Prof. Alessandro Bazzi, Referee, University of Bologna and IEIIT-CNR
Prof. Matteo Sereno, Referee, University of Torino

Politecnico di Torino
2018

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Ahsan Mahmood
2018

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my loving parents and my lovely daughters.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisors Prof. Paolo Giaccone, Prof. Carla-Fabiana Chiasserini and Prof. Jerome Härri for their kind support and guidance throughout my PhD study and research. I really appreciate their patience, encouragement and immense knowledge contributing towards the successful completion of my degree. Indeed, it was a great opportunity to work with them, thus, allowing me to gain valuable experience.

I would also like to thank Prof. Claudio Casetti for his insightful comments and contributions in this work.

I acknowledge the funding from EURECOM (France) that made it possible for me to carry out my research.

Lastly, I would like to thank my family: my parents, siblings and my wife, for being supportive and helpful throughout my studies.

Abstract

Content caching will be an essential feature in the next generations of cellular networks. Indeed, a network equipped with caching capabilities allows users to retrieve content with reduced access delays and consequently reduces the traffic passing through the network backhaul. However, the deployment of the caching nodes in the network is hindered by the following two challenges. First, the storage space of a cache is limited as well as expensive. So, it is not possible to store in the cache every content that can be possibly requested by the user. This calls for efficient techniques to determine the contents that must be stored in the cache. Second, efficient ways are needed to implement and control the caching node. In this thesis, we investigate caching techniques focussing to address the above-mentioned challenges, so that the overall system performance is increased.

In order to tackle the challenge of the limited storage capacity, smart proactive caching strategies are needed. In the context of vehicular users served by edge nodes, we believe a caching strategy should be adapted to the mobility characteristics of the cars. In this regard, we propose a scheme called RICH (RoadsIde CacHe), which optimally caches content at the edge nodes where connected vehicles require it most. In particular, our scheme is designed to ensure in-order delivery of content chunks to end users. Unlike blind popularity decisions, the probabilistic caching used by RICH considers vehicular trajectory predictions as well as content service time by edge nodes. We evaluate our approach on realistic mobility datasets against a popularity-based edge approach called POP, and a mobility-aware caching strategy known as netPredict. In terms of content availability, our RICH edge caching scheme provides an enhancement of up to 33% and 190% when compared with netPredict and POP respectively. At the same time, the backhaul penalty bandwidth is reduced by a factor ranging between 57% and 70%.

Caching node is an also a key component in Named Data Networking (NDN) that is an innovative paradigm to provide content based services in future networks. As compared to legacy networks, naming of network packets and in-network caching of content make NDN more feasible for content dissemination. However, the implementation of NDN requires drastic changes to the existing network infrastructure. One feasible approach is to use Software Defined Networking (SDN), according to which the control of the network is delegated to a centralized controller, which configures the forwarding data plane. This approach leads to large signaling overhead as well as large end-to-end (e2e) delays. In order to overcome these issues, in this work, we provide an efficient way to implement and control the NDN node. We propose to enable NDN using a stateful data plane in the SDN network. In particular, we realize the functionality of an NDN node using a stateful SDN switch attached with a local cache for content storage, and use OpenState to implement such an approach. In our solution, no involvement of the controller is required once the OpenState switch has been configured. We benchmark the performance of our solution against the traditional SDN approach considering several relevant metrics. Experimental results highlight the benefits of a stateful approach and of our implementation, which avoids signaling overhead and significantly reduces e2e delays.

Contents

Chapter 1

Introduction

The next generations of cellular networks will be able to support wide ranging applications and large number of connected devices. In order to meet the ever increasing demand of data traffic passing through the network as well as to fulfil stringent performance requirements, new technologies will be implemented and smart techniques will be introduced for managing the network and providing content to users. In this regard, content caching is one of such techniques that will be used to fulfil the above challenges. In this work, we investigate caching techniques which include decision of contents to be placed inside cache and management of the cache nodes. In this chapter, first we give an overview of 5G, an upcoming generation of cellular network, followed by the role of caching in future networks and then we introduce the caching techniques investigated in this dissertation.

1.1 An outline of 5G networks

Here we give an overview of a next generation cellular network, i.e., 5G network. 5G standard is business driven. Differently from 4G, 5G systems integrate key market verticals in the standardization process. The use cases originating from these vertical sectors drives the 5G requirements [1]. Some of the business cases in each vertical sector are given in Table 1.1.

The analyses of the uses cases in these vertical industries derive the key performance parameters in 5G. Some of the key performance indicators are data rate, latency, coverage, density etc. In terms of data rate, the most demanding use cases

Table 1.1 Vertical industries and associated business cases in 5G [2]

Vertical Industry	Business cases
eHealth	Assets management in hospitals, remote monitoring of health
Factories-of-the-future	Process optimization inside factory, remote maintenance
Energy	Grid access, grid backhaul, grid backbone
Automotive	Automated driving, road safety and traffic efficiency services
Media & Entertainment	Ultra high fidelity media, on-site live event experience, collaborative gaming

are from the Media & Entertainment vertical industry, where high quality video services require high downlink data rates and sharing of user generated contents require high uplink data rates. This can be attributed to immensely changing habits and expectations of users regarding media consumption. User experience is broadening as there are varied types of user devices, e.g., smartphones, virtual reality devices, tablets and wearables, and users prefer to remain connected all the time; not only at home but also on the roads.

For supporting the use cases in Media & Entertainment vertical, caching capabilities will be included in the 5G network [3]. Other than this vertical, the benefits of caching can also be reaped in the automotive vertical industry for real-time streaming of content, thus enabling futuristic features like autonomous driving and supporting traffic safety applications.

In order to implement a 5G network, the following new technologies will emerge:
Software Defined Networking (SDN): It is an innovative networking paradigm that split the control and the data planes of the network. It allows us to realize a flexible network where network devices are controlled through a logically centralized SDN controller. As a consequence, SDN enables a programmable network, lowers operational costs and allows for flexible services [4].
Network Functions Virtualization (NFV): It is a new technique to provide network functions such as firewall, intrusion detection system, caching, etc. NFV decouples such functions from dedicated hardware appliances, hence allowing them to run in software on commercial servers.
Cloud Radio Access Network (C-RAN): It is a cloud-computing based architecture for future cellular networks and also known as centralized radio access network. In this architecture, all computational resources of base stations are combined in a central pool, which receives radio signals gathered from radio units via optical transmission network. C-RAN allows to implement a flexible network able to support

multiple technologies and lowers capital and operational costs.
Mobile Edge Computing (MEC): It enables computing and storage capabilities at the edge of the cellular network. By running an application in close proximity to the user, end-user experience is improved and network congestion is reduced.

Among the above mentioned building blocks of 5G network, we are particularly interested in SDN that can be used to enable efficient content caching solutions.

1.2 Role of caching in future networks

The significance of content caching in a network is undoubtedly large. The network nodes equipped with caches give the possibility to store in advance contents required by the users. In this way, the content requests may be satisfied from an intermediate point in the network instead of reaching distant servers. This significantly benefits both users and network operators. From the user's perspective, the ability to retrieve content from intermediate nodes in the network, reduces delays and enhances the quality of experience. From the operator's point of view, the network overhead is greatly reduced, especially if multiple users request the same content (e.g., popular videos and live sport streams). In light of such advantages, future 5G systems will integrate caching capabilities in the network, especially to provide media and entertainment services [3].

Content caching is also a key component in Named Data Networking (NDN) [5] which is an innovative paradigm to provide content based services in future networks. NDN is gaining importance due to a dominant use of the Internet as a content distribution network; there is an increasing usage of content-based applications such as video sharing, social media networking, and e-commerce. As compared to legacy networks, naming of network packets and in-network caching of content make NDN more feasible for content dissemination.

Apart from the substantial benefits that can be possibly gained by including content caching capabilities in the networks, there are a few challenges for the deployment of the caching nodes. First of all, the size of a cache is limited and it incurs a cost as well. So, it is not possible to store in the cache every content that can be possibly requested by user. This calls for efficient techniques to determine the contents that must be stored in the cache. In addition, it is also important to

determine efficient ways to implement and control a cache node so that it is able to function effectively. In this thesis, we investigate the caching techniques that can help to store such contents so that the overall system performance is increased.

1.3 Caching techniques

In the thesis, we investigate the caching techniques and address the following questions.

- *What* to cache? It determines which contents are stored in cache as defined by caching policies.
- *How* to control cache node? It determines how a cache node carries out its functionality.

1.3.1 What to cache

In order to determine which contents get stored in cache, there are two types of caching policies.

- Cache insertion: such policies determine which contents are inserted in the cache. They can be further categorized into *standard caching* and *prefetching*. Web caching and caching in NDN nodes are examples of the former category where the contents requested by users are usually stored in the cache. While, in the latter category, the contents are proactively stored in the cache before users request them and are similar to Content Distribution Network (CDN).
- Cache eviction: such policies determine which content to evict when the cache becomes full. According to the eviction policy used in this work, the contents removed at higher priority are the ones that have the lowest probability to be downloaded, e.g., contents that have already been delivered to users.

In this work, we investigate prefetching policies to proactively insert contents in cache, with particular focus on vehicular users. We consider an urban scenario where network coverage is provided by Edge Nodes (ENs) (e.g., cellular base station,

access point, roadside unit), which are equipped with cache and vehicular users download content by connecting to ENs. In the vehicular environment, in addition to the limited storage capacity available in cache, high mobility of cars makes it even more challenging to store the contents efficiently. Depending upon speed, traffic conditions and roadside signals at intersections, cars spend varying amount of time under coverage of an EN. Clearly, amount of content delivered to a user depends on the coverage time spent under the EN. Therefore, we propose to exploit mobility information while devising a prefetching policy for vehicular users. Note that this is in accordance with the current trend in 5G systems [6], which foresees a dynamic caching system to prefetch content in the ENs, based on future demand estimation obtained by user's context information such as direction and speed. So, we classify the prefetching policies in vehicular environment in the following two categories.

- Mobility-agnostic: policies that do not consider mobility information for prefetching contents in the EN. For example, storing the most popular contents in the cache.
- Mobility-aware: policies that take into account mobility information for prefetching contents.

As mentioned above, the mobility-aware prefetching policies consider the time spent by a car under the coverage of an EN, which is termed as dwell time. The information about the dwell time of car can be very useful to prefetch content in EN. Consider a simple scenario that a car traverses a sequence of two ENs to download a content. If the entry time of the car in the first EN and its dwell time under each EN are known in advance, then by assuming a fixed value of the data rate, we can accurately determine part of the content downloaded from each of the two ENs. So, only the needed parts of the content can be prefetched in the caches at the ENs.

In practical scenario, it is not possible to accurately know the dwell times of the car at the traversing ENs. Therefore, we propose a probabilistic approach, where, based on the past measurements, probability distribution of the dwell time is computed at each EN. If a car passes through a sequence of ENs and requests for content as it enters the coverage area of the first EN, then assuming to know only the sequence of the ENs our approach utilizes the dwell time distributions to calculate the probability of downloading contents while under the coverage of each EN along the path. Based on that, our caching scheme is able to compute the contents required

to be prefetched at each EN. The contents are prefetched before the arrival of the car under any EN using some mobility prediction that is out of scope of this work. Since the actual dwell time of the car at any EN is unknown, the uncertainty exists regarding the time at which the car leaves the coverage of any EN. As the caching scheme is run when the car enters the first EN of the sequence, such uncertainty is lesser in the first EN as compared to that in the subsequent EN. In general, the uncertainty increases for the ENs appearing later in the sequence. The consequence of this uncertainty is that the prefetched chunks at the ENs may not be sufficient. In order to cope with this situation, few of the contents stored in two consecutive ENs (of the sequence) are duplicated. Clearly, the number of duplicated contents in two consecutive ENs appearing later in the sequence will be larger.

In our work, we show that the knowledge of the dwell time distribution can be used to make efficient prefetching decisions. We run simulations to support our claim and compare the numerical results with two state-of-art prefetching policies.

1.3.2 How to control cache node

Content caching is an integral component of NDN where each network node is equipped with a cache. As a result, a content request can be satisfied from any node in the network. In NDN, the contents are not prefetched, instead they are inserted in the cache as requested by users. Although NDN is more suitable for content dissemination as compared to legacy networks, its implementation requires drastic changes in the network. One feasible solution is to take the standard SDN approach and implement an NDN node as a cache-equipped SDN switch controlled by the SDN controller. In this solution, a local cache is attached with an SDN switch for content storage, and the control of the cache-equipped SDN switch, i.e., cache node, is offloaded in the controller. Therefore, the cache node completely relies on the controller for its functionality, rendering it incapable to make any decision by itself. The cache node does not maintain any state inside the switch resulting in a *stateless* data plane. In this solution, the over dependence of the cache-equipped switch on the SDN controller results in a large amount of signaling overhead, especially when we have a network of cache-equipped switches and all of them rely on the controller for their operations. Moreover, the need to communicate with the controller for all decisions also increases the end-to-end delay at the user end.

We propose that the control of the cache node should be transferred from the controller to the switch, by maintaining local states inside the switch. So, we have *stateful* data plane implementation of the NDN node. As a result, the cache-equipped switch can function at its own by making decisions based on switch local knowledge, thus eliminating the need to interact with the controller after the initial configuration. We use OpenState [7] to implement such an approach. We evaluate performance of our solution considering relevant metrics and benchmark it against the traditional SDN approach as well.

1.3.3 Cache node in vehicular network

So far, we have described the control of the cache node in NDN. It is important to discuss it in the context of vehicular network, where, the EN serves as the cache node as described in Section 1.3.1. The main functionalities of the EN are to store contents and to interact with users to deliver contents. In order to realize such operations, in addition to the cache, the EN should also be equipped with processing capabilities. In this way, the EN implements the Internet protocol suite and becomes able to provide content streaming service to users.

The provision of large amount of processing capabilities at the ENs may not be feasible. For example, access points or roadside units, being resource constrained, must be attached with a server for enabling them to provide the content streaming service. A feasible solution seems that we should take the SDN approach and offload all the processing requirements of the ENs to the SDN controller. In this way, the EN becomes just a cache-equipped SDN switch (with a radio interface) that relies completely on the controller to realize its functionality. As described above, following the traditional SDN approach leads to a large amount of control traffic exchanged between the SDN controller and the switches, as well as large end-to-end delays. As similar in the case of the NDN node, we envision that the EN should be able to function at its own by following the stateful SDN approach as well. This can significantly reduce the control traffic and the latency to retrieve content.

1.4 Structure of thesis

Chapter 2 initially presents the network architecture and the associated protocols for content caching in vehicular network. Furthermore, it describes our prefetching policy that uses the mobility information of the cars to determine the contents to store in the caches located at the network edge. Then, it provides simulation methodology as well as the numerical results to evaluate the performance of our system and shows its effectiveness against other state-of-art prefetching policies. Chapter 3 presents a practical approach to implement the cache node in NDN using stateful SDN. Moreover, it provides the implementation of the NDN node using OpenState. Then, it discusses the performance evaluation of our solution considering relevant metrics as well as benchmark our approach against traditional SDN implementation. Finally, the conclusions of the thesis are drawn in Chapter 4.

Chapter 2

The RICH Prefetching Policy

2.1 Motivation

Connected cars are considered by drivers as a projection of their homes on the road, and the same seamless connectivity and content-based services are expected on the road as well. Communication of connected cars with the network infrastructure can support a wide variety of applications, ranging from critical, safety-related applications to the provision of entertainment services (e.g., video distribution). Given the latency-sensitive nature of these applications, a deployment of servers and content at the network edge, hence close to vehicular users, is desirable. Such a requirement, together with the expected increase in the number of connected cars, calls for a significant redesign of the network architecture in order to support high-performance connectivity and cloud-based content and applications.

As shown by a recent study on caching architectures [8], placing content caches at network edge performs remarkably well when compared to the centralized caching approach, mainly because the latency is reduced. Additionally, from the operator's perspective, core network congestion is reduced and thus the performance perceived by users is higher. Mobile Edge Computing (MEC), considered as one of the key technologies for 5G networks, provides computing and storage platforms at the edge of the mobile network. MEC can therefore be used to push content to edge nodes, in close proximity to connected cars.

A major limitation of this approach is that edge nodes do not have the same storage flexibility as the cloud, and efficient strategies have to be developed to store

the right content at an Edge Node (EN) (e.g., cellular base station, AP, roadside unit) and provide it to users under its coverage. Also, the thirst for wireless capacity has led to a reduced coverage size of ENs, which requires content to be replicated in multiple ENs to meet user demands. The design of caching policies has therefore been widely investigated in the past [9–13].

Yet, connected cars add further challenges to edge caching strategies. They are, after all, highly-mobile vehicles and such a mobility requires storage strategies to be optimized not with respect to the current content popularity, rather to the expected content popularity among future users about to enter the coverage of ENs. Furthermore, the dynamics of connected cars augmented by the limited coverage size of ENs require content to be stored where connected cars have a chance to actually download it, say in slow-speed areas, such as congested intersections. All these aspects create a challenging triumvirate for edge caching for connected cars: limited coverage, low storage capacity and high mobility. Tackling such a triumvirate requires edge caching strategies to be adapted to car mobility and connectivity.

In the past, mobility-based caching policies have been investigated, such as in [14, 15], but they require a full knowledge of the trajectory of each car, rising concerns about drivers' privacy. Also, most of caching policies (as the ones in [9, 16, 17]) are not tailored to in-sequence delivery of content, typically required by future on-board streaming applications. A major design challenge for caching policies is therefore to rely only on coarse mobility information (sequences of waypoints, dwell time, etc.), while supporting in-sequence content delivery.

In this work, we propose RICH (RoadsIde CacHe), a prefetching policy for edge caching specifically adapted to highly dynamic environments with a coarse knowledge of car trajectories. We consider an urban environment where cars can connect to ENs, in order to download content as shown in Fig. 2.1. Our approach is based on the knowledge of the sequence of travel waypoints, and some aggregate statistics about the distribution of the dwell time under the coverage of each EN. Note that this is in accordance with the current trend in 5G systems [6], which foresees a dynamic caching system to prefetch content in the ENs, based on future demand estimation obtained by users' context information, such as direction and speed.

With respect to classical works on caching, the peculiarities of the scenario addressed in our work are two. First, we consider a data streaming application in which the content is divided into fixed-size chunks, strongly correlating the

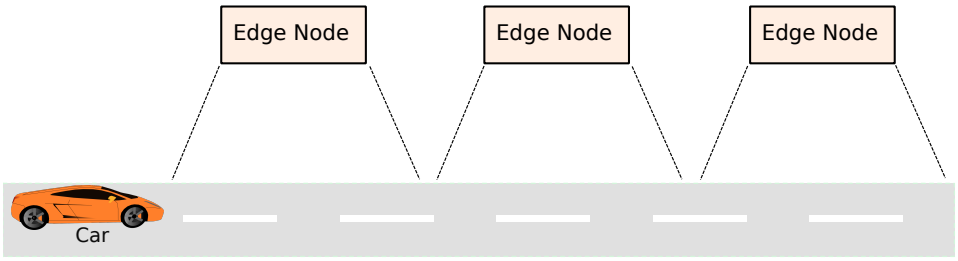


Fig. 2.1 Scenario: a car traversing multiple Edge Nodes

download process at each EN, due to the in-sequence chunk delivery. Second, mobility introduces another level of correlation in the request process among different ENs. We remark that the instantaneous popularity of a chunk at an EN depends on the actual temporal and spatial trajectory of all cars interested in the corresponding content. The goal of our strategy is to cache in advance the chunks in the sequence of ENs traversed by the car, by choosing the chunks that will be most likely downloaded at each specific EN. This increases the *cache hit probability*, i.e., the probability that the content chunks are downloaded directly from the cache, which in turn greatly reduces the backhaul traffic and content access delay.

Our novel contributions are as follows:

- *Architecture and protocol:* we introduce a split content caching architecture, with a centralized module located in the backhaul which is responsible to manage the content stored in the caches located on ENs. This centralized approach is amenable to being realized through the Software Defined Networking (SDN) paradigm. We also define the system protocol governing the interaction between various entities in the vehicular network scenario.
- *Analytical formulation:* we describe an analytical model capable of predicting the probability of downloading a chunk of a content from a given EN.
- *Caching scheme:* we leverage the previous model to develop a mobility-aware prefetching strategy that selects what (i.e., which chunk) and where (i.e., in which ENs) to cache, based on the distribution of the dwell times.
- *Implementation:* we evaluate our solution under a realistic urban traffic dataset of the city of Bologna. Our simulation model is comprehensive and it closely

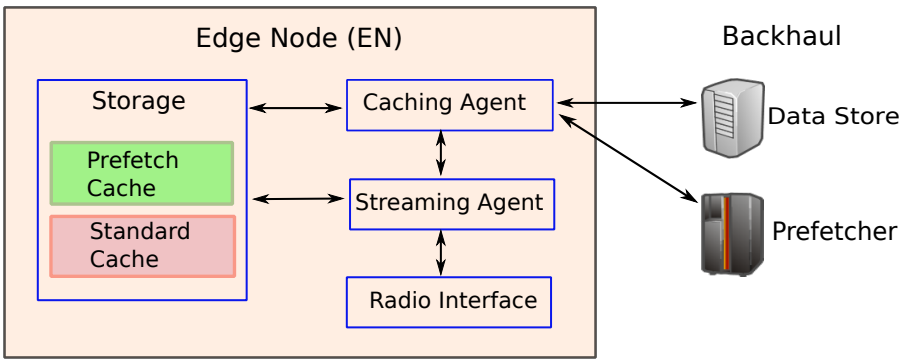


Fig. 2.2 Network caching architecture

mimics a real scenario. We compare our proposed RICH scheme with two state-of-art prefetching policies: pure popularity-based caching (POP) and a mobility-aware caching strategy called netPredict [14].

The rest of the chapter is organized as follows. In Sec. 2.2, we describe our network architecture and the system protocol. The proposed RICH edge caching strategy is explained in Sec. 2.3. In order to compare the performance of the RICH approach with alternative policies, we explain the simulation scenario and the adopted methodology in Sec. 2.4, while Sec. 2.5 reports the simulation results. We discuss related work in Sec. 2.6. Finally, in Sec. 2.7 we draw our conclusions.

2.2 Network Architecture

We consider an urban scenario in which vehicular users traverse coverage areas of several ENs, where each EN is responsible for the communication in a specific geographical area. Due to the dynamic nature of the vehicular network, along with a large number of users and a volatile wireless channel, users downloading data may experience severe service disruptions. Our goal is therefore to design a caching architecture that provides users with a timely delivery of the content, while also reducing the backhaul bandwidth consumption.

We focus on a challenging application such as content streaming, for which in-order delivery should be ensured. We let ENs cache parts of the content requested

by the vehicular users, and we envision the presence of a *Prefetcher* module that instructs, in advance, each EN on which part of the content to store. In this way, the content portion that each user is expected to download from an EN, will be readily available for delivery at the EN when actually reached by the user.

Specifically, in the proposed architecture shown in Fig. 2.2, the Prefetcher is located in the network backhaul, along with a Data Store module. Both of them are connected with all the ENs. The Data Store provides a catalog for all the contents, each of which is assumed to be composed of a sequence of *chunks*, each univocally identified. The Prefetcher can acquire (e.g., through the user's navigation system) or predict the sequence of ENs that the user will traverse in the near future. Also, thanks to past measurements, the distribution of the cars' dwell time under each EN is known to the Prefetcher, as well as the available storage in the caches of the ENs and of the users' content requests. Based on such information, the Prefetcher defines *which chunks* of *which content* should be cached in advance at *which* EN, and instructs the ENs accordingly. Upon getting the instructions from the Prefetcher, each EN retrieves the needed chunks from the Data Store before arrival of the user in the coverage area. This allows leveraging non-real-time data transfer and thus relaxes the QoS requirements for the content transfer into the caches.

Each EN is responsible for providing streaming service to the user, and, in order to provide timely content delivery, it utilizes a local caching storage, divided in two parts:

- the *Prefetch Cache*, which contains the chunks prefetched from the Data Store, as per Prefetcher's instructions;
- the *Standard Cache*, which stores chunks that were not available in the Prefetch Cache at the time the user requested them, and they have been retrieved from the Data Store.

Notably, the latter cache mimics the behavior of a standard caching system, where a new chunk is stored just after a cache miss is experienced. On the contrary, the Prefetch Cache behaves similarly to a content delivery network (CDN) node, where the content is proactively stored into the node.

Moreover, the EN carries out its tasks by means of two agents:

- the *Streaming Agent* reads the chunks from the cache and streams them to the user; one separate stream is managed for each user under coverage;
- the *Caching Agent* interacts with the Prefetcher, fetches chunks from the Data Store, inserts the prefetched/missed chunks into the cache, and manages the cache according to the adopted eviction policy.

2.2.1 System protocol

The Streaming Agent contacts the Caching Agent whenever a new content request arrives, as well as whenever a requested chunk is not available in the cache. The Caching Agent, in turn, informs the Streaming Agent as the needed chunk becomes available in the cache. The interactions between the aforementioned entities are further clarified by the space-time diagrams in Figs. 2.3 and 2.4, which depict the protocol followed, respectively, when a user starts requesting a content and while the car connects with the subsequent ENs to complete the content download, if necessary. In both cases, the proposed protocol is composed of two phases:

- *prefetching phase:* an EN prefetches the chunks from the Data Store according to the Prefetcher's instructions, and the EN serves the user only on the basis of the chunks stored in the Prefetch Cache;
- *data recovery phase:* the EN fetches the missing chunks from the Data Store and transmits the chunks in sequence using both the Prefetch Cache and the Standard Cache, if needed.

In both figures, chunks downloaded from the Prefetch Cache are depicted in green, whereas the ones downloaded from the Standard Cache are depicted in red. All the messages corresponding to control information are depicted in blue.

We assume, for simplicity, that a generic content c is divided into equal sized chunks, and we define $\mathscr{K}_c$ as the set of all the corresponding chunk identifiers. Let $k \in \mathscr{K}_c$ be a generic chunk identifier of content c and $d_{c,k}$ denotes the corresponding actual data. Let v be a generic user and/or its car.

In more details, Fig. 2.3 refers to a scenario in which car v enters the coverage area of the first EN along its path and and the user requests the first chunk k_1 of content c. At the EN, the request is forwarded to the Prefetcher. Actually, the

real implementation of such step is more complex since the request is sent to the Streaming Agent, which registers a new stream for the user and, in turn, forwards the chunk request to the Prefetcher through the Caching Agent. In the following we will omit such level of details in the interaction among the modules internal to the EN. According to our assumptions, the Prefetcher knows the sequence of ENs the car will traverse. Based on such information, it runs the prefetching policy (described in Sec. 2.3) and instructs the first EN to prefetch the set $\mathscr{K}_1 \subseteq \mathscr{K}_c$ of chunks of content c. The EN, through the Caching Agent, checks which of the required chunks are already available in the Prefetch Cache and sends a message to the Data Store asking for the missing ones, represented by $\mathscr{K}_2 \subseteq \mathscr{K}_1$. The Data Store sequentially sends the chunks data to the EN, i.e. all the chunks $[d_{c,k}]_{k\in\mathscr{K}_2}$. The Caching Agent inserts the received chunks in the Prefetch Cache and informs the Streaming Agent about the data availability. Then the Streaming Agent initiates the data stream towards the user, sending the chunks in sequence.

If a chunk is not available in the Prefetch Cache, the system enters the *data recovery phase*, since the data must be retrieved directly from the data store. The latter situation may happen in two cases: (i) some prefetched chunks have been evicted from the cache to make space for other chunks; (ii) all the prefetched chunks have been transmitted and the car is still under coverage. In both cases, the Streaming Agent attempts to serve the car by getting the chunks from the Standard Cache. Otherwise, the missing chunks, denoted by $\mathscr{K}_3 \subseteq \mathscr{K}_c$, are fetched from the Data Store for transmission to the user. $\mathscr{K}_3$ is chosen large enough to compensate for the round-trip time from the Prefetch Cache to the Data Store and to guarantee a continuous streaming from the Data Store to the user, using the EN as relay until the car leaves the coverage area.

Fig. 2.4 shows only the prefetching phase occurring at the second EN and at the subsequent ENs traversed by the car. The eventual data recovery phase is not shown since it is identical to the one in Fig. 2.3. When the car enters the coverage of the first EN, the Prefetcher instructs all the subsequent ENs to prefetch the required chunks before the arrival of the car in their coverage areas. As shown in Fig. 2.4, the EN receives a message to prefetch the chunks $\mathscr{K}_5 \subseteq \mathscr{K}_c$. For the subset $\mathscr{K}_6$ of chunks in $\mathscr{K}_5$ that are not yet available in the cache, the EN asks the Data Store and stores the corresponding chunks in the Prefetch Cache.

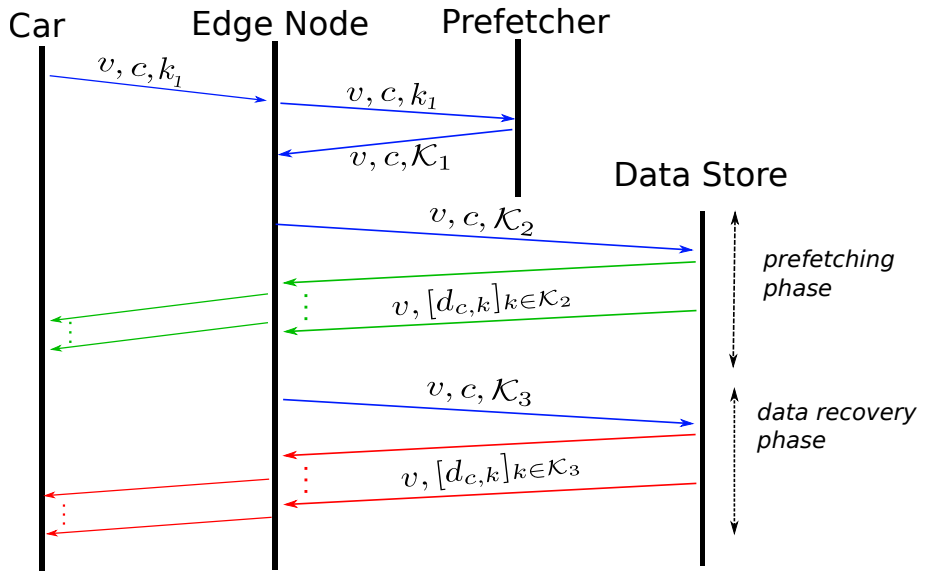


Fig. 2.3 Protocol followed by the first EN traversed by a car.

When the car enters the EN coverage, it sends a chunk request for the first missing chunk k_2. The EN forwards this message to the Prefetcher in order to notify it about the car entrance in the coverage area. The EN is responsible to send all the required chunks to the car which are available in the local cache. Eventually, the Prefetcher may trigger a data recovery phase in order to compensate for missing chunks in the cache.

2.3 RICH prefetching policy

Given the caching architecture discussed above, we propose an efficient strategy for the Prefetcher to select the chunks to store in the Prefetch Cache of the ENs along the path in advance. We recall that the car path is provided in terms of the sequence of traversed ENs and the distribution of the dwell time under the ENs is obtained through historical data. By prefetching chunks in the cache, the user can experience higher throughput whenever the chunks are locally downloaded from the EN and not from the backhaul, while leading to better utilization of backhaul network resources.

The section is organized as follows. As first step, we motivate our approach considering a toy-case example and highlight the benefits of knowing the statistical

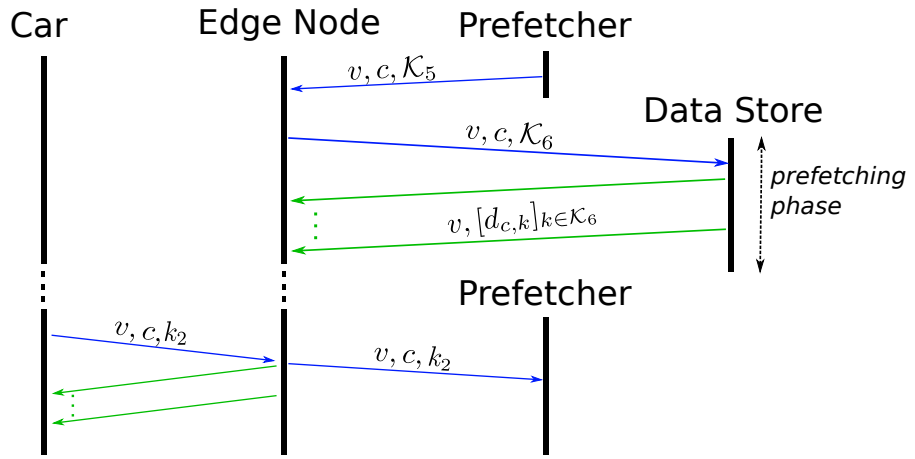


Fig. 2.4 Protocol followed by the ENs after the first EN traversed by the car.

distribution of the dwell time experienced by the cars under each EN. As second step, we theoretically evaluate the probability that a specific chunk is downloaded from an EN by a tagged car, which will be used to define our prefetching policy. Indeed, our approach consists in letting each EN store those chunks whose probability to be downloaded by a car is above a given threshold.

In the following, for simplicity, we will refer always to the Prefetch Cache as "cache".

2.3.1 A toy-case example

Consider a single EN covering the area of an intersection controlled by a traffic light. The dwell time of each car and, thus, the number of downloaded chunks depends on the traffic light state. Let us now focus on the subset of cars that request a specific content starting from its first chunk. Assume that 80% of these cars experience green light, thus the corresponding dwell time under the EN is small and the users (labelled by "fast") can download up to 10 chunks. The 20% of these cars experience red, thus the corresponding dwell time under the EN is large and the users (labelled by "slow") can download up to 100 chunks. Hence, the average number of chunks that can be downloaded by a car is 28.

Assume now that the system adopts a prefetching policy that takes its decisions based on the average number of chunks downloaded by users, as, e.g., the state-of-art netPredict policy discussed later in Sec. 2.4.3. Such a policy will store just the first 28 chunks of the content in the cache. This can be seen as an inefficient decision for both kinds of users. Indeed, for the fast users, able to download just the first 10 chunks, the additional 18 chunks stored in the cache are completely useless and a waste in terms of cache occupancy. For the slow users, able to download up to 100 chunks, the cache is heavily underutilized since only the first 28 chunks are available.

In our work we propose instead to exploit the distribution of the dwell time under each EN to optimize the performance of the overall caching system. For the above toy example, two other possible solutions can be easily envisaged. If storing just the first 10 chunks, no "waste" of cache occupancy will be experienced by the fast users, whereas the slow users will keep experiencing cache misses as in the netPredict case. But the reduction in cache occupancy can be exploited by storing other contents. If instead we store the first 100 chunks, all the users will find the chunks in the cache, at the cost of higher cache occupancy.

Note that the possible inefficiencies in a single cache scenario is exacerbated when considering a sequence of caches, due to the required in-sequence delivery of the chunks. Thus, we advocate the use of prefetching policies that are aware of the distribution of the dwell time, and not only of the average dwell time under an EN, as netPredict does.

2.3.2 Chunk download probability for large caches

We now derive the probability that a user successfully downloads a chunk from the cache. Let us focus on one car and one specific content that a user wishes to download from the traversed ENs. Assume that EN i, with $1 \leq i \leq N$, is the ith EN along the path of the car, composed of N ENs.

We start by defining the chunk delivery process under a best-case scenario in which all ENs cache the whole content, i.e., any chunk can be available at any EN along the path. This implicit assumption of very large cache is aimed at devising a simple model that will be tailored to small caches later in this section.

Let Y_i be the random variable representing the last chunk received from EN $i \geq 1$, and let $Y_0 = 0$ by definition. Then the set of chunks downloaded from EN i is given

by: $\{k|k \in (Y_{i-1}, Y_i]\}$. Thus the probability that chunk k is downloaded from EN $i \geq 1$ is, for any $k \in \{1, \ldots, K\}$,

$$\phi_i(k) = \mathbb{P}(k \in (Y_{i-1}, Y_i]) = \mathbb{P}(Y_i \geq k \ \wedge \ Y_{i-1} < k) \tag{2.1}$$

Note that if N is large enough, the content will be downloaded for sure from some EN, thus $\sum_{i=1}^{\infty} \phi_i(k) = 1$.

Let X_i be the random variable representing the total number of chunks downloaded from EN i by the user. The probability density function (pdf) of X_i depends mainly on two factors: (1) the mobility of the car, since, e.g., longer dwell times under the EN coverage typically imply larger amounts of download data, and (2) the actual throughput obtained by the user when connected to the EN, which in turn depends on the wireless data rate and on channel contention among other users. In Sec. 2.4.1, we will describe how to compute the pdf of X_i in the urban scenario under study.

Based on our definitions, it is easy to see that for $i \geq 1$,

$$Y_i = Y_{i-1} + X_i = \sum_{j=1}^{i} X_j \,. \tag{2.2}$$

The following theorem relates the download probability $\phi_i(k)$ to X_i.

Theorem 1. *Given a car traversing a sequence of ENs, the probability to download a specific chunk k from EN i, with* $1 \leq i \leq N$*, can be expressed as*

$$\phi_i(k) = \sum_{n=1}^{k-1} \mathbb{P}(X_i \geq k - n | Y_{i-1} = n)\, \mathbb{P}(Y_{i-1} = n) \,. \tag{2.3}$$

Proof. Given (2.2), we can write (2.1), for any $i \geq 1$, as:

$$\begin{aligned} \phi_i(k) &= \mathbb{P}(Y_{i-1} + X_i \geq k \ \wedge \ Y_{i-1} < k) \\ &= \sum_{n=1}^{k-1} \mathbb{P}(X_i \geq k - Y_{i-1} | Y_{i-1} = n)\, \mathbb{P}(Y_{i-1} = n) \\ &= \sum_{n=1}^{k-1} \mathbb{P}(X_i \geq k - n | Y_{i-1} = n)\, \mathbb{P}(Y_{i-1} = n) \,. \end{aligned}$$

□

Note that, as expected, for $i = 1$ the expression in (2.3) becomes $\phi_1(k) = \mathbb{P}(X_1 \geq k)$ since the user will download chunk k from the first EN only if the total amount of downloaded chunks from EN 1 is greater than k. Now, thanks to the well-known property of the expectation of non-negative integer random variables, we can claim:

Property 1. $\sum_{k=1}^{K} \phi_i(k) = \mathbb{E}[X_i]$.

The following corollary holds in the special case when the car dwell times under the ENs are i.i.d. random variables. Note that this case is addressed here for completeness, as well as to provide a more explicit expression of the ϕ's, however our approach does not require such an assumption.

Corollary 1. *Let the random variables X_i's be i.i.d. and defined on a positive support. Let $f_X(k)$ be their discrete pdf. Then, for any $i \geq 1$, we have:*

$$\phi_i(k) = (f_X * \phi_{i-1})(k) \tag{2.4}$$

where $*$ is the convolution operator.

Proof. When X_i's are i.i.d.,

$$\phi_i(k) = \sum_{n=1}^{k-1} \mathbb{P}(X \geq k-n)\, \mathbb{P}(Y_{i-1} = n).$$

Thus, (2.3) can be rewritten as:

$$\begin{aligned}
\phi_i(k) = & \sum_{n=1}^{k-1} \mathbb{P}(X \geq k-n) \sum_{t=1}^{n} \mathbb{P}(X = t | Y_{i-2} = n-t) \cdot \\
& \mathbb{P}(Y_{i-2} = n-t)) \\
\overset{z=n-t}{=} & \sum_{z=1}^{k-1} \mathbb{P}(X \geq (k-t)-z) \cdot \\
& \sum_{t=1}^{k-1} \mathbb{P}(X = t | Y_{i-2} = z) \mathbb{P}(Y_{i-2} = z) \\
= & \sum_{t=1}^{k-1} \mathbb{P}(X = t) \cdot \\
& \sum_{z=1}^{k-1} \mathbb{P}(X \geq (k-t)-z) \mathbb{P}(Y_{i-2} = z) \\
= & (f_X * \phi_{i-1})(k) . \qquad (2.5)
\end{aligned}$$

□

The complexity of computing the convolution for all N ENs is $O(N^2K^2)$. Notably, such computation can be done offline, based on past statistics.

We now consider the case of an EN cache of limited size. Let M_i be the available space in terms of chunks at EN i. Then we can introduce a discrete random variable, $\widehat{X}_i$, which represents the total number of cached chunks downloaded from EN i by the user, given the available room in the cache of EN i. The pdf of $\widehat{X}_i$ is given by:

$$\mathbb{P}\left(\widehat{X}_i = x | M_i\right) = \begin{cases} \mathbb{P}(X_i = x) & 0 \leq x < M_i \\ \mathbb{P}(X_i \geq x) & x = M_i \\ 0 & x > M_i . \end{cases} \qquad (2.6)$$

Indeed, it is not possible to download more than M_i cached chunks, and the events corresponding to a number of downloaded chunks larger than M_i in the original model with large cache size, now correspond to downloading all the M_i available chunks.

Given the above distribution, the probabilities $\phi_i(k)$ can be computed as in (2.3), or when $\widehat{X}_i$'s are i.i.d. as in (2.5) .

To further clarify the behavior of the download probability at each EN, we present an example below.

Example 1: Consider a toy scenario in which a car traverses four ENs and large-sized caches are available. X_i's are i.i.d. with a symmetric triangular distribution and mean value equal to 10 chunks, i.e., the average number of chunks downloaded at each EN is 10. Fig. 2.5 shows the download probabilities $\phi_i(k)$ at each EN i for each chunk k, computed by applying the results of Corollary 1. From Fig. 2.5 we can observe that, at the first EN, $\phi_1(k)$ decreases as k increases since the randomness in the mobility reduces the probability of downloading farther chunks. Due to the limited support of the distribution of X_1, $\phi_1(k)$ becomes zero for $k \geq 20$ chunks. At the second EN, $\phi_2(k)$ is now bell-shaped, since values of k close to zero correspond to the case in which X_1 takes very small values (which is unlikely), i.e., the car speed is very high under the coverage of the first EN, hence the user does not have enough time to download any chunk. The maximum is obtained around 15, which is reasonable since in the case of deterministic mobility with $X_i = 10$ chunks for any i, the chunks to be downloaded would be exactly in the interval $[10, 20]$, which is symmetric around 15. The chunk download probability from the following ENs ($i > 2$) still exhibits a bell-shaped behavior, but with an expanded support. This is due to the larger uncertainty on downloading a specific chunk from a given EN, which, in turn, is due to the increased randomness in the number of previously delivered chunks.

2.3.3 The RICH prefetching algorithm

Based on the previous definition of the chunk download probability, we define our prefetching policy that determines which ENs should store each chunk of the content, taking into account the required sequence of chunks to be downloaded for each user, obliviously of the content popularity. The goal of our scheme is to ensure that the probability with which a user can download a chunk from any of the ENs is greater than a given threshold τ, with $\tau \in [0, 1]$. Our scheme thus identifies the *smallest* set of ENs that should cache each chunk so that its download probability exceeds τ. This set depends on the combined probabilities of downloading the chunk from

Algorithm 1 RICH prefetch with a single-threshold

Require: τ ▷ Probability threshold
Require: $\{\phi_i(k)\}_{i,k}$ ▷ Download probability for any chunk and for any EN
1: **for** $k = 1, \ldots, K$ **do** ▷ For any chunk k
2: $\mathscr{S}(k) = \emptyset$, $p_k = 0$ ▷ Init
3: $\mathscr{F}(k) \leftarrow \{\phi_i(k) | \phi_i(k) > 0\}_i$ ▷ Store download prob. in descending order for chunk k
4: **while** $\mathscr{F}(k) \neq \emptyset$ and $p_k \leq \tau$ **do** ▷ Check threshold
5: $p_{top} \leftarrow$ remove the highest prob. from $\mathscr{F}(k)$
6: $i_{top} \leftarrow$ EN index corresponding to p_{top}
7: $p_k = p_k + p_{top}$ ▷ Accumulate the probabilities
8: $\mathscr{S}(k) = \mathscr{S}(k) \cup \{i_{top}\}$ ▷ Update the list of ENs
9: **end while**
10: **end for**
11: **return** $\{\mathscr{S}(k)\}_k$ ▷ ENs where to prefetch each chunk
12: **return** p_k ▷ Final download probability

each EN. Thus, setting τ allows to control the maximum amount of chunks to be downloaded from the backhaul.

The pseudocode of RICH is reported in Algorithm 1, which, for each chunk k, returns the set of ENs, $\mathscr{S}(k)$, that must store k according to the RICH prefetch policy. After the initialization, for each chunk k (lines 1-10), RICH considers the set $\mathscr{F}(k)$ of corresponding download probabilities (line 3). To avoid degenerate solutions, RICH just considers the ENs for which it is possible to download chunk k, i.e. $\phi_i(k) > 0$. Now the algorithm iterates (lines 4-9) on all the download probabilities in decreasing order (line 5). Chunk k is now stored at the EN i corresponding to the highest $\phi_i(k)$ value. If this value is already greater than τ, no other EN should cache k. Otherwise, the EN corresponding to the second top value of $\phi_i(k)$ should store the chunk too. Eventually, chunk k will be cached at as many ENs, associated to the top $\phi_i(k)$ values, as necessary so that the sum of their $\phi_i(k)$ exceeds τ. Notably, the probabilities can sum up since referring to disjoint events. The set $\mathscr{S}(k)$ contains the list of all the EN to store content k and it is updated in line 8.

At the end of procedure, p_k represents the estimated download probability of chunk k from *any* EN in $\mathscr{S}(k)$, thus with $|\{\mathscr{S}(k)\}|$ copies of chunk k. Whenever $p_k \geq \tau$, the prefetching procedure has run successfully. Otherwise, it fails since it is not possible to find any set of ENs where to prefect the chunk in order to satisfy τ. In such a case, we do not store any copy of the chunks. Note the last ENs of

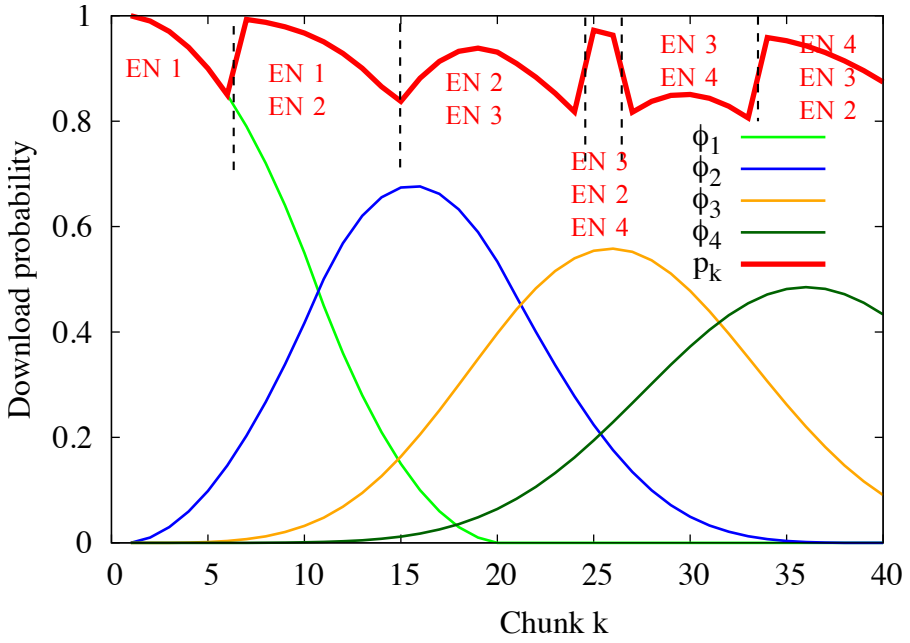


Fig. 2.5 Download probability for each EN and overall download probability, p_k, for the RICH caching policy with a single threshold $\tau = 0.8$, given a toy example scenario with $\mathbb{E}[X_i] = 10$ chunks.

a path experience a significant *border effect* since the policy is not able to exploit subsequent ENs where to prefetch the chunks. In Algorithm 1, the computation complexity of each loop is $O(N \log N + N)$, due to the sorting in line 3 and the fact that each chunk can be stored in at most N ENs. Thus the overall complexity is $O(KN \log N)$.

To understand the effect of setting τ, we consider some extreme cases. If we set $\tau = 0$, a single copy of each chunk is stored in the most probable EN. Whereas, if we set $\tau = 1$ and the threshold is reached, the chunk is stored in any EN for which the download is possible; thus, the maximum number of copies is stored. Otherwise, if the threshold is not reached, then the chunk is not stored in any EN. The value of τ can be numerically optimized to maximize a given performance metric, i.e. the overall cache hit probability.

In the example of Fig. 2.5, we show the different $\phi_i(k)$ corresponding to a simple triangular distribution for X_i with $E[X_i] = 10$ chunks. We also show the final p_k obtained with RICH by setting $\tau = 0.8$ and (in red) the EN (or ENs) storing chunk

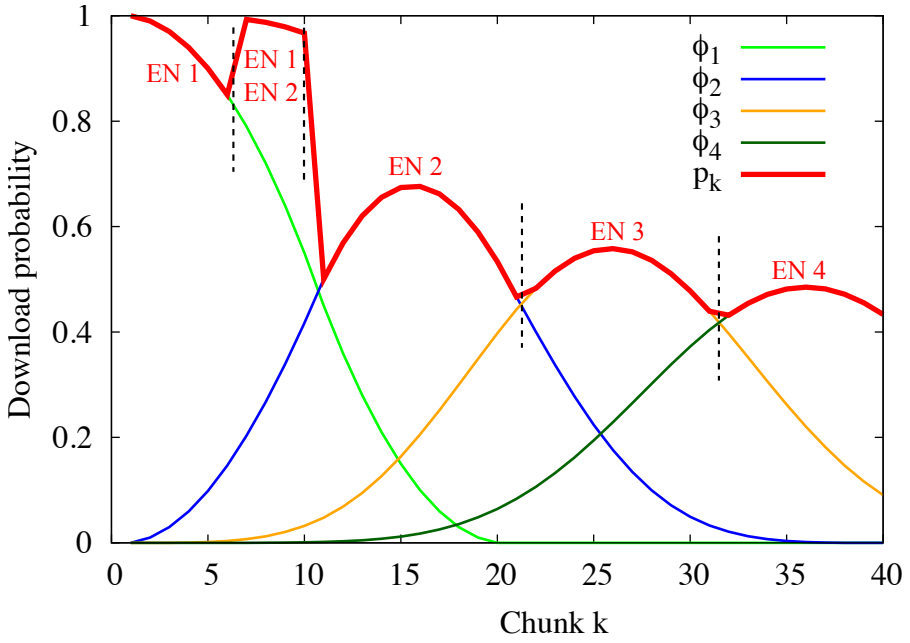


Fig. 2.6 Download probability for each EN and overall download probability, p_k, for the RICH caching policy with multiple thresholds, being $\tau_1 = 0.8$ (for $k \in [1, 10]$) and $\tau_2 = \tau_3 = \tau_4 = 0.4$ (for $k > 10$).

k. The non-monotonic behavior of p_k is due to the different number of copies that are stored at the ENs for different chunks, as shown by the curve labeled “single-th" in Fig. 2.7. The behavior of the number of copies is due to two different effects. Intuitively, as k increases, the uncertainty about the possibility to download chunk k increases, thus the RICH caching algorithm compensates it by creating a higher number of copies. Indeed, in Fig. 2.7 the number of copies grows from 1 to 3. At the same time, the uncertainty is usually larger for the chunks which are “between” two ENs and thus the number of copies can be non-monotonic, as shown around chunk 26 in Fig. 2.7. To understand this second effect, consider the example shown in Fig. 2.8, depicting $\phi_i(k)$ and $\phi_{i+1}(k)$ for two subsequent ENs. The most probable chunks at each EN (whose download probability is greater than τ) will be stored as just one copy, whereas the other chunks (between the most probable chunks) will be stored as two copies, due to the uncertainty in the precise moment when the car will enter EN $i+1$.

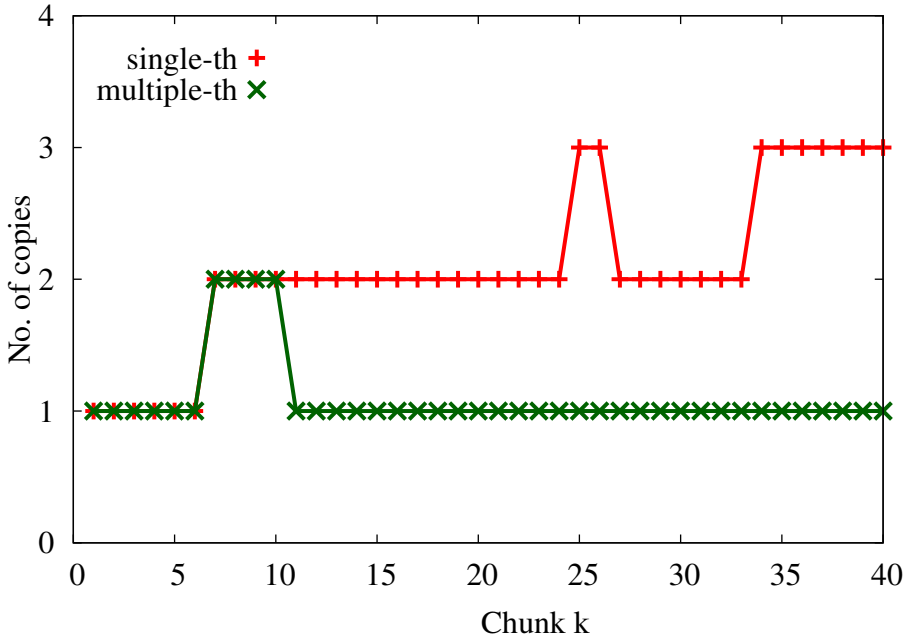


Fig. 2.7 Number of copies for the single-threshold and multiple-thresholds RICH prefetching scheme.

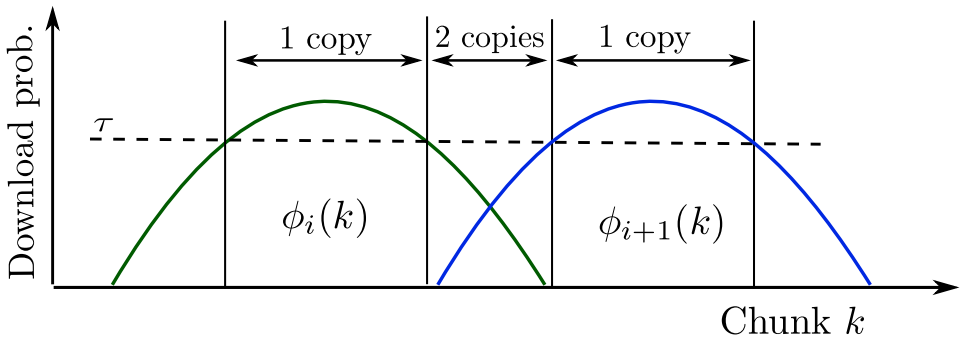


Fig. 2.8 Example of number of copies for two subsequent ENs.

Coping with uncertainty

We now propose two further enhancements to our scheme, in order to efficiently cope with chunk requests that are farther in the future.

Multi-threshold RICH policy. Our intuition is that a fixed threshold τ, as the one used in Algorithm 1, may be unfit for chunks that are expected to be downloaded in farther ENs, for which the level of uncertainness is larger. Indeed, for such chunks

the single-threshold RICH policy will naturally require a very large number of copies, thus wasting precious space in the cache that could be better used for other users. We can therefore consider a dedicated threshold τ_i for each EN i, leading to a decreasing sequence of thresholds $\{\tau_i\}_i$. The actual values of the thresholds can be optimized numerically to maximize the cache hit probability. We define instead the range of chunks for which threshold τ_i is adopted for EN i as the set of all the chunks such that $\phi_i(k) \geq \phi_j(k)$, for any other EN $j \neq i$. The reason for this choice is that the chunks with the largest download probability at one EN must affect the actual threshold value to use on such EN. Importantly, the complexity of optimizing the thresholds grows as N, i.e., the number of ENs in the path, and not as the number of chunks in a content (which may be arbitrary large, depending on the specific kind of content).

Fig. 2.6 shows the same scenario of Fig. 2.5 but using different values of threshold for the four ENs. Only $\tau_1 = 0.8$ and all the other thresholds are equal to 0.4. The final p_k is lower with respect to the single threshold case for $k > 10$, but this is due to the smaller number of copies (usually 1 for most of the chunks) as shown in Fig. 2.7. This reflects the intuition that for the chunks whose uncertainty is large, it is better not to "waste" cache storage with multiple copies.

Refreshing the policy. As already observed, due to the high level of uncertainness, for "far-in-the-future" chunks RICH will compensate with a large number of copies, wasting a large amount of cache storage while achieving a marginal performance improvement. Thus, we design RICH in order to prefetch the content just on the initial sequence of ENs and to refresh the prefetching decision before entering the first EN not considered in the previous prefetching decision. This approach allows to trade the effectiveness of the prefetching decision with the allowed time to prefetch the content.

Eviction policy

Consider now a generic scenario with multiple users served by one EN. Since the EN cache is finite, an eviction policy is needed to determine which chunks should be removed when the cache is full and new chunks must be inserted. We adopt the following eviction policy: the chunks removed at higher priority are the ones that have been already delivered, i.e., the chunks destined to users not anymore under coverage. Among these chunks, our policy evicts with higher priority the chunks

with the lowest download probability. In this way, we can efficiently exploit the cache under chunk demands that vary both in space and in time.

2.4 Simulation scenario and methodology

In order to investigate the performance of RICH, in this section we start by introducing the scenario and the real-world vehicular traces utilized in our work. Then we explain the adopted methodology used for the comparison of the RICH scheme against state-of-art alternative solutions, in terms of some relevant performance metrics.

2.4.1 Reference scenario

We consider a realistic urban environment. We take as reference scenario a 2 km $\times$ 2 km urban area of the Italian city of Bologna, illustrated in Fig. 2.9. The detailed vehicular mobility traces were obtained to reproduce the experimental data obtained through the traffic detectors available at the intersections, as described in [18]. The total trace duration is 79 minutes comprising 11,079 vehicles (approximately, 950 are simultaneously on the map) and representing 120 minutes of the morning rush hours, under quite stationary traffic conditions.

In this urban section, we select the major traffic arteries and assume to place eight ENs along them, as represented in Fig. 2.9 in correspondence of main intersections regulated by a traffic light. Let $\mathscr{R}$={A,B,C,D,E,F,G,H} be the set of the ENs. All ENs have the same cache size. We assume an ideal radio range of 100 m at each EN, thus the coverage areas are not overlapping. The circles in Fig. 2.9 show the actual coverage area of each EN.

In order to investigate the effect of the prefetching policy across multiple ENs, we consider only those cars in the trace that enter the coverage of any three ENs. Among the large number of the possible combinations of the three ENs, we consider just the ones with a large number of users and denote them as *significant paths*. In total we identify 7 significant paths with a minimum number of cars equal to 45 each. The overall number of cars following the significant paths are 2,053, and our investigation focuses on such subset of cars. Table 2.3 reports the number of cars in

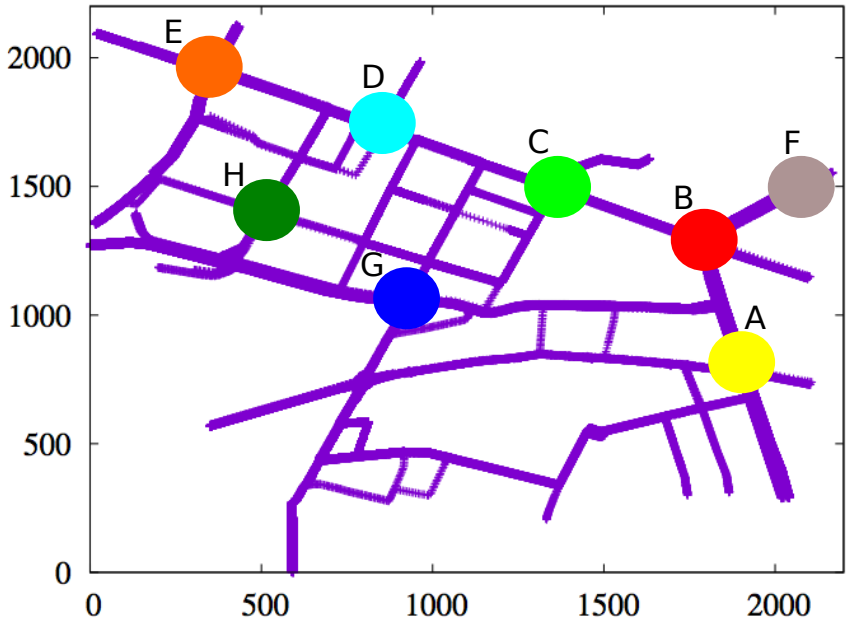


Fig. 2.9 Reference urban area in the city of Bologna. Circles represent EN locations and the corresponding coverage areas. Distances are expressed in meters.

each of the significant paths. Given the vehicular traces of all the cars traversing the significant paths, for each EN we derive the empirical distribution of the car dwell times, needed by the Prefetcher.

Finally, although RICH can be implemented on any Infrastructure-to-Vehicle technology (ITS-G5/DSRC, WiFi or LTE), we opted for the IEEE 802.11a WiFi standard, yet operating on the 5.9 GHz frequency band. This choice has been made over, e.g., ITS-G5 [19], for the following reasons. First, since our application is not strictly a safety-related application, we would not use ITS-G5 to transmit on the ITS-G5 bands (at least 5.875-5.905 GHz). Nevertheless, we want to avoid transmitting on the WiFi bands at 5.4 GHz so as to not interfere with non-vehicular communications and increase the capacity available for content streaming delivery. Second, the ETSI is preparing a deregulation of the ITS band to allow WiFi access, e.g., 802.11a, under a 'detect-and-avoid' principle against ITS-G5 technology [20, 21]. Thus, according to C-ITS [22], we test our RICH prefetching on the Service Channel 1 (SCH 1) at 5.875-5.885 GHz and, accordingly, ignore potential coexistence with ITS-G5.

Table 2.1 OMNeT++ simulation parameters

Parameter	Value	Parameter	Value
WiFi	802.11a	WiFi active scan	false
Frequency band	5 GHz	Beacon interval	100 ms
Bandwidth	10 MHz	Receiver sensitivity	-85 dBm
Max bitrate	54 Mbps	SNIR threshold	4 dB
Rate control	AARF	Background noise power	-101 dBm
MTU	1 kbytes	Pathloss type	Free space
Tx power	33 dBm	Pathloss coefficient	3
Antenna gain	1 dB	Max communication range	110 m
Retry limit	7	Max interference range	200 m

2.4.2 Simulation methodology

We investigate the performance of RICH by simulating the vehicular traffic according to the mobility trace of Bologna, focusing on the most significant routes. We developed a discrete-event simulator, based on OMNeT++ [23], which models the network architecture in Fig. 2.2 with 8 deployed ENs, as shown in Fig. 2.9. Even if in our model we implemented both the Prefetch and the Standard Caches, in the following we will investigate just the performance due to the Prefetch Cache, denoted as "cache". We assume an ideal communication link from the ENs to the Prefetcher, with a propagation delay equal to 10 μs, corresponding to a Prefetcher located physically in the same urban area. Instead, the Data Store is connected to the ENs, with a 100 Mbps communication link and a propagation delay of 2 ms. We model the wireless access network with a detailed 802.11a[1] model provided in Inet-Framework 3.3 [24]. The simulation parameters are defined in Table 2.1.

[1]The implementation of 802.11a in OMNeT++ simulator contains only one transmission queue at MAC layer, which is not suitable for high throughput applications. The queue can be filled with data of a single car, hence starving out other cars under coverage. In order to address this issue, we have implemented per-destination queuing at MAC layer. There is a dedicated per-destination queue (PDQ) for each car which is served using round robin scheduling algorithm. Also, there is an additional dedicated queue for storing broadcast frames (e.g. beacons). This broadcast queue has priority over all other per-destination queues. After a transmission finishes, broadcast queue is checked if it contains any packets to be served. After that, PDQs are served according to round robin method. As a result, the EN was able to fairly serve all the cars under its coverage. Finally, when a car departs the coverage area, the packet in its corresponding PDQ is purged

Table 2.2 Number of cars under each EN in Bologna urban area

EN	A	B	C	D	E	F	G	H
Avg. cars	27.51	13.36	7.88	4.52	8.35	6.02	3.42	2.79
Total cars	1580	1510	527	382	337	1456	161	206

Table 2.3 Significant paths identified in Bologna urban area

Path	ABC	ABF	CDE	FBA	FBC	HDC	HGA
No. of cars	54	555	337	810	91	45	161

In our reference scenario, each significant path is based on 3 ENs, we run the prefetching policy and evaluate the performance at the first two ENs, to avoid the border effects due to the last EN, as described in Sec. 2.3.3. Since E appears in the significant paths only as third EN, according to Table 2.3, we exclude it in the following evaluations, thus in our scenario we have just 7 effective ENs.

Table 2.2 shows the statistics about the temporal average of the number of users under each EN (given that at least one car is present), and the total number of distinct cars under coverage. EN A is located at the most crowded intersection, thus we expect heavy congestion and lower per-user throughput. On the other hand, H is at the least crowded intersection, for which we expect the least number of users on average, thus high per-user throughput.

To evaluate the download probability for any chunk and for any EN, i.e. $\{\phi_i(k)\}_{i,k}$, we must determine the number of downloaded chunks X_i at each EN. Let W_i be the random variable of the dwell time of a generic car under EN i. Let b be the total bandwidth available when a user is under coverage of a EN. Let u be the average number of cars under coverage. Let s be the total size in bits of each chunk. We can claim approximatively:

$$X_i = \frac{W_i \cdot b}{s \cdot u_i} \tag{2.7}$$

Substituting the empirical values of W_i and u_i (given in Table 2.2) in (2.7), we estimate the distribution of X_i and then the empirical $\phi_i(k)$ for any EN i and chunk k. Finally, we evaluate $\widehat{X}_i$ to take into account the finite cache, by applying (2.6).

Each time a new car enters the coverage of any EN for the first time, it generates a content request according to Zipf's distribution with exponent $\alpha = 0.75$, coherently with the value observed in [25]. The size of one content is 2,600 chunks, with each chunk being 65 kbytes large. The normalized cache size is defined as the cache size divided by the total size of all contents in the catalog. The size of the catalog is 10 contents.

The performance metrics we consider are as follows:

- *cache throughput*, measured in bit/s: average amount of data received by the user over time and directly downloaded from the cache, i.e., due to a cache hit;
- *cache hit probability*, $P_{hit} \in [0,1]$: fraction of chunk requests that are satisfied by directly downloading the chunk from the cache, i.e. in the event of a cache hit;
- *backhaul traffic*, measured in bit/s: average amount of data downloaded from the server over time, i.e. due to the event of a cache miss;
- *normalized backhaul overhead*: total amount of additional traffic transferred between the DataStore and the ENs with respect to amount of traffic delivered to the users, normalized by the traffic delivered to the users; it can be also negative in case of *content reuse* in the cache, i.e. whenever a user finds in the cache a content that was prefetched for other users;
- *normalized cache size*, $\hat{C} \in [0,1]$: size (i.e. maximum allowed occupancy) of each cache, normalized by the catalog size;
- *network cache occupancy* $\in [0,N]$: average total cache occupancy among the N ENs normalized by the catalog size.

Notably, the interval of time considered in the above statistics takes into account just the total time under which a car is under coverage of any EN.

2.4.3 Alternative prefetching policies

In our simulations, we compare the performance of RICH to that of two different prefetching approaches.

The first one is POP, a mobility-agnostic approach, which prefetches only the most popular contents in the cache, starting from the most popular one up to the storage saturation. POP requires knowledge of the popularity level of all the contents in advance but it is provably optimal in terms of hit probability for a single cache and under a stationary content request process [25]. We tailor the behavior of POP for our data streaming scenario, in which each content is divided into chunks. POP stores the chunks of the most popular content items in sequence; if no space is available in the cache for the whole content, only the initial chunks of the content are stored, until the cache is full.

The second prefetching approach is *netPredict*, proposed in [14] and discussed in Sec. 2.6. netPredict exploits both spatial and temporal predictions of the car path based on the previous history, and, thus, the cars' dwell time under the coverage of the ENs. For a fair comparison with RICH, we assume that the spatial prediction in netPredict is perfect and, hence, the sequence of ENs traversed by the car is known in advance. Based on the knowledge of the *average* dwell time under each EN and the value of the average bandwidth available at each EN, netPredict stores a number of chunks given by their product. Thus, the adopted communication model in [14] is identical to (2.7), but exploits just $E[X_i]$ instead of the distribution of X_i as in RICH. In a nutshell, netPredict can be seen as a special case of RICH for deterministic X_i.

2.4.4 Statistical analysis of the reference scenario

In our reference scenario explained in Sec. 2.4.1, the ENs are deployed at road intersections where vehicular traffic is regulated using traffic lights. Cars may arrive under the coverage of the ENs following various paths, as shown in Table 2.3. Also, the number of cars on each path is different. Considering such traffic conditions, the dwell time distributions (and hence the distributions of $\widehat{X}_i$) of the cars at the ENs show interesting statistical properties. The statistical description of $\widehat{X}_i$ is reported in Table 2.4, for a cache size of 2,600 chunks. We observe that, if incoming cars at some EN i take different paths (with a significant number of cars in each path), the distribution of $\widehat{X}_i$ shows high value of skewness and kurtosis. For example, Table 2.3 shows that EN B is involved in several paths; cars enter B's coverage with different incoming and outgoing roads. Therefore, the distribution of $\widehat{X}_B$ shows the highest value of skewness and kurtosis. A similar behavior can be observed in case of ENs A, D and H. On the other hand, the incoming cars under the coverage of E and G

Table 2.4 Statistical description of empirical $\widehat{X}_i$ in Bologna urban area.

EN	A	B	C	D	E	F	G	H
Skewness	2.11	2.48	0.41	1.74	-0.11	-0.15	-0.18	1.62
Kurtosis	8.57	11.60	2.71	6.89	2.63	2.22	1.68	5.98

follow only the paths CDE and HGA, respectively. Hence, the distributions of $\widehat{X}_E$ and $\widehat{X}_G$ show lower values of skewness and kurtosis.

2.5 Numerical results

Considering the simulation scenario and the adopted methodology explained in Sec. 2.4, we compare the performance of RICH with that of POP and netPredict policies. The comparison is carried out both at the network edge and the backend. Furthermore, we evaluate the measure of benefit for the user as well as the operator using the utility functions. Finally, we show gain in the performance by considering more accurate car mobility information.

Firstly, for each of the significant paths mentioned in Table 2.3, we present the chunk download probability $\phi_i(k)$ for chunk k and every EN i in that path. These download probabilities are shown in Figs. 2.10–2.16. It can be observed from the figures that the download probability for the initial chunks is 1 in the first EN of every path due to no uncertainness in the mobility information. Whereas, the level of uncertainness increases in the second and the third ENs, and consequently, the bell-curves are more spread at the farther ENs.

We use $\phi_i(k)$ to compute the multi-threshold RICH policy for each significant path. We have optimized the thresholds τ_i adopted in our RICH policy. We used an exhaustive approach to find the combination of the thresholds at the three ENs for maximizing the cache hit probability. Table 2.5 shows the optimal thresholds in our scenario for different cache sizes, assumed fixed among all ENs. In general, the optimal threshold value τ_1 in the first EN is higher as compared to τ_2 and τ_3. This is because there is less uncertainty regarding the mobility of the cars in the first hop. So the initial chunks can be downloaded from the first EN with high probability, while

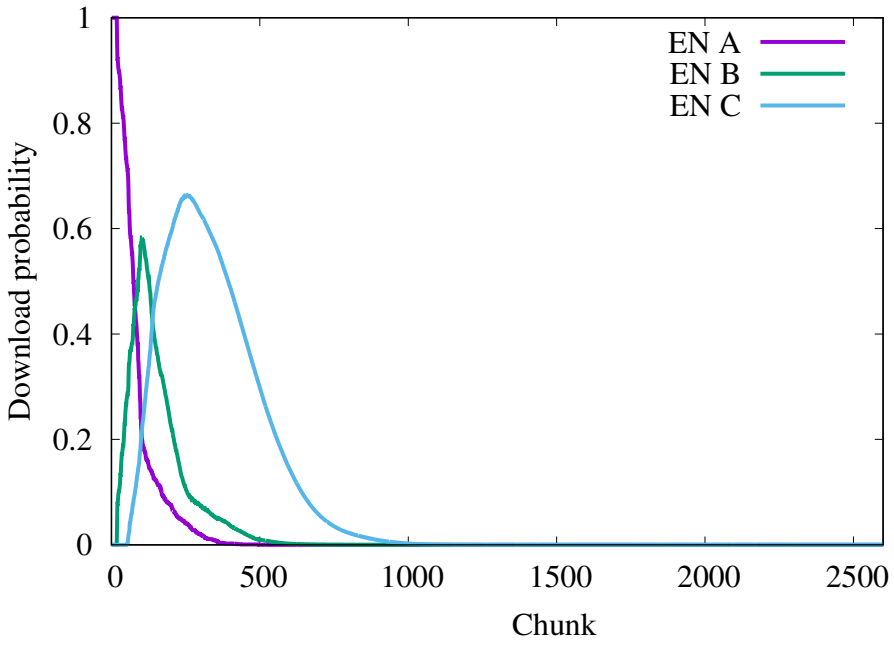


Fig. 2.10 Chunk download probability at each EN of the path=ABC in the urban Bologna scenario.

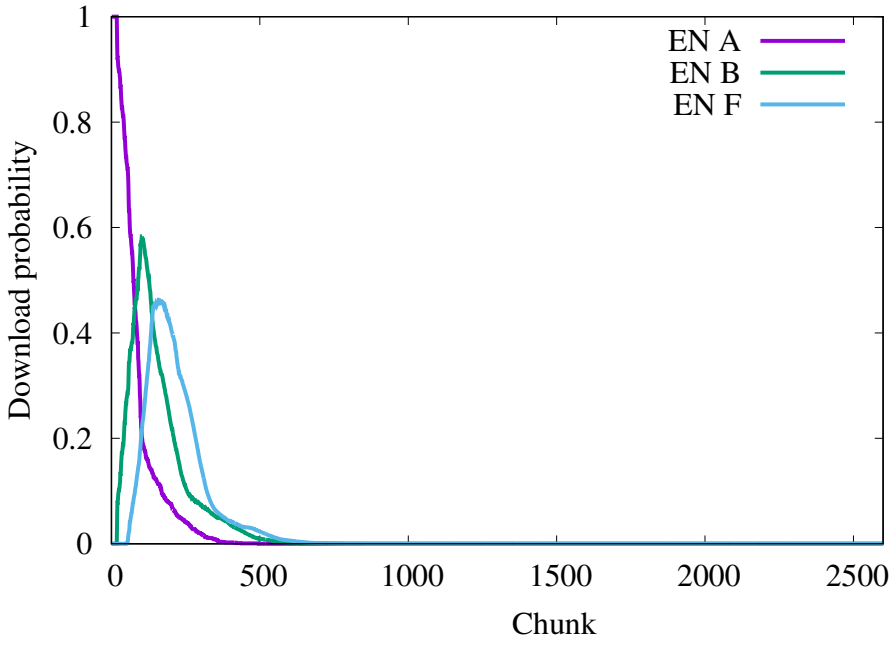


Fig. 2.11 Chunk download probability at each EN of the path=ABF in the urban Bologna scenario.

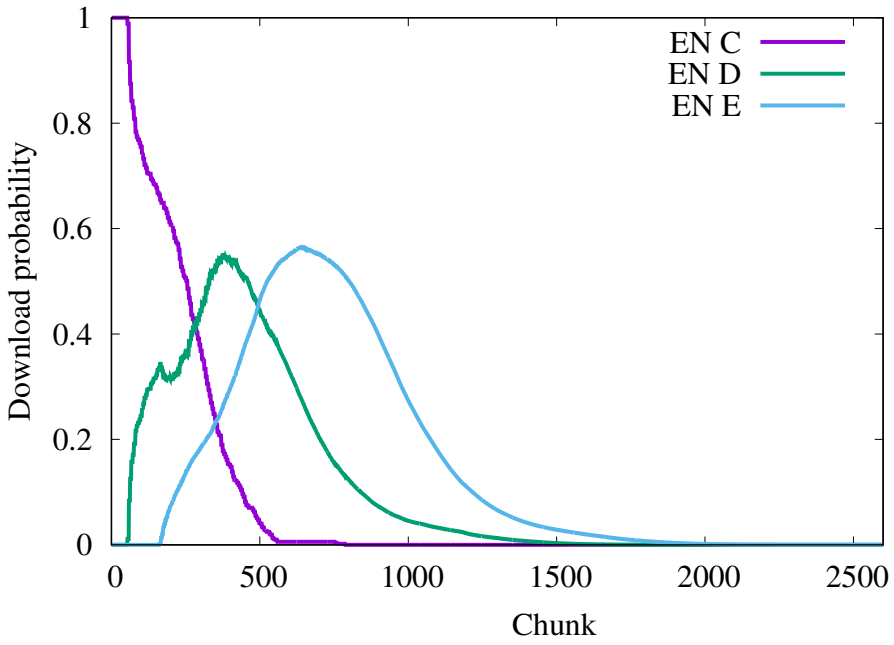


Fig. 2.12 Chunk download probability at each EN of the path=CDE in the urban Bologna scenario.

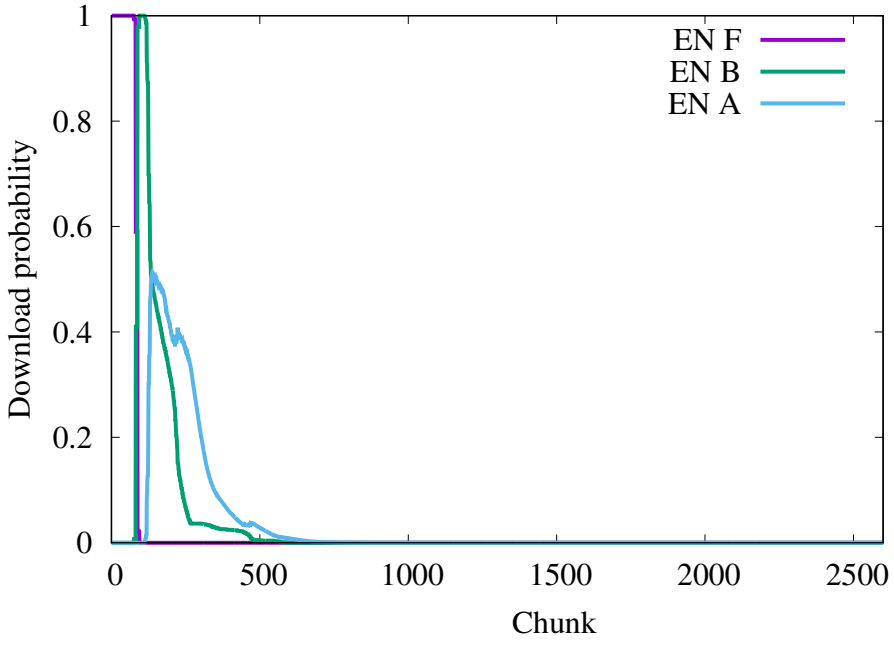


Fig. 2.13 Chunk download probability at each EN of the path=FBA in the urban Bologna scenario.

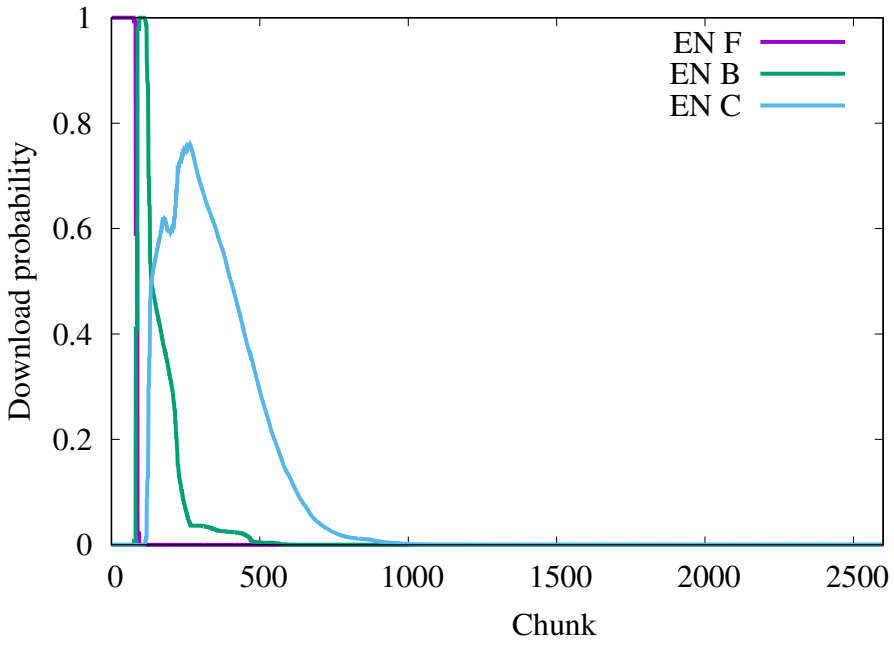


Fig. 2.14 Chunk download probability at each EN of the path=FBC in the urban Bologna scenario.

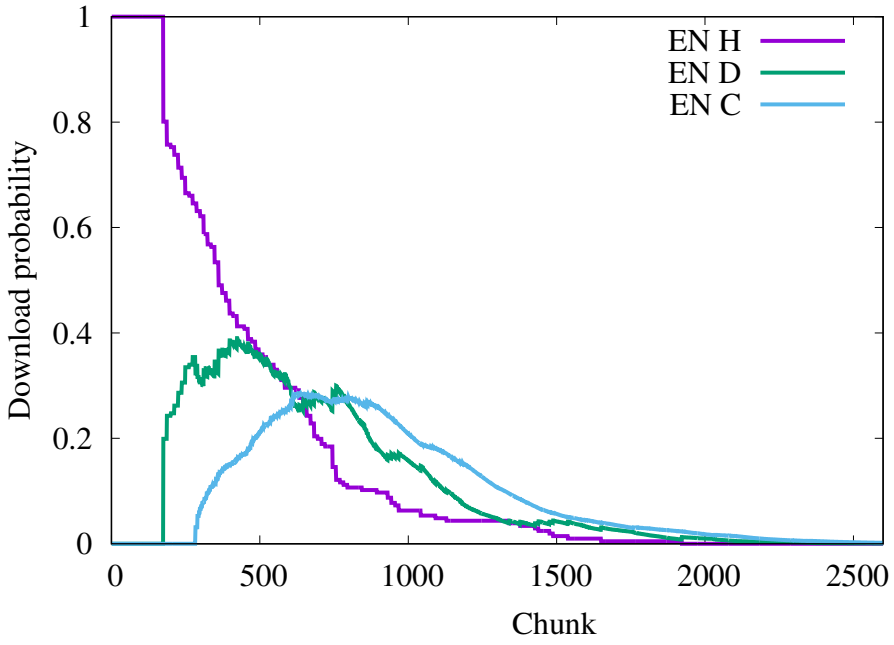


Fig. 2.15 Chunk download probability at each EN of the path=HDC in the urban Bologna scenario.

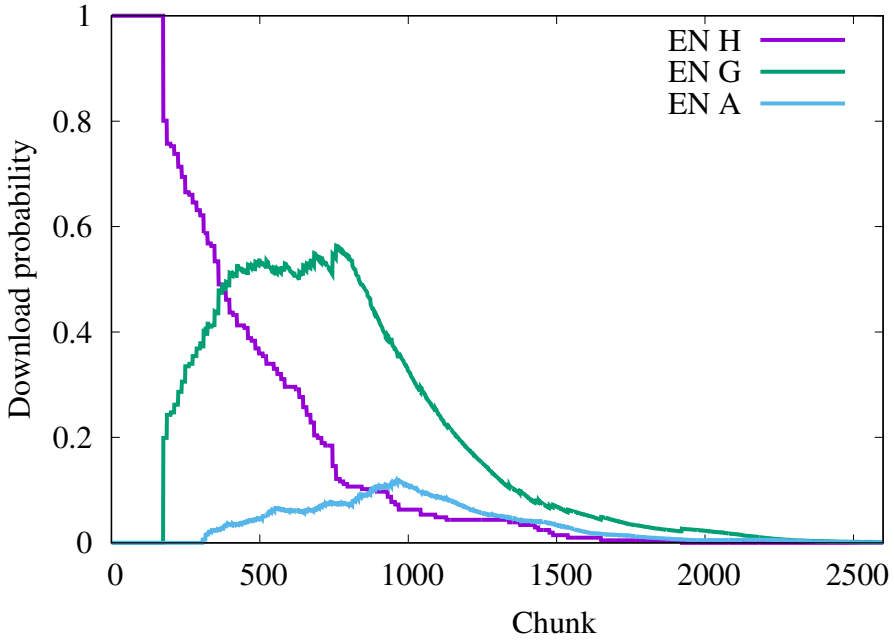


Fig. 2.16 Chunk download probability at each EN of the path=HGA in the urban Bologna scenario.

farther chunks are less likely to be downloaded due to increasing randomness in the mobility.

We evaluate the performance only in the first two ENs, to avoid the the border effect due to EN 3, as discussed in Section 2.3.3. By construction, τ_3 depends on the chunks that are most probably downloaded from EN 3. However, such chunks can also be downloaded from EN 2. The higher value of τ_3 as compared with τ_2 allows some additional chunks to be stored in EN 2 which increases the cache hit probability. For cache size ≥ 7800 chunks, the optimal value of the thresholds does not change, because of the large cache capacity, not fully utilized.

Note that the threshold optimization using the exhaustive approach utilizes the same threshold values for all the EN sequences/paths due to scalability issues. However, their optimal values depend on the specific EN sequence and could be different for the various paths. The future work will include an algorithm to optimize the threshold values separately for each path, consequently, the cache hit probability can be enhanced even further.

Edge network performance. First, we evaluate the performance of the three policies at the network edge since it directly impacts the content access delay.

Table 2.5 Optimal thresholds for different cache sizes

CACHE SIZE		OPTIMAL THRESHOLD		
#contents	#chunks	τ_1	τ_2	τ_3
1	2600	0.88	0.67	0.70
2	5200	0.92	0.58	0.67
3	7800	0.99	0.57	0.67
4	10400	0.99	0.57	0.67
5	11000	0.99	0.57	0.67

Fig. 2.17 compares the cache hit probability as a function of the normalized cache size $\hat{C}$, for POP, netPredict and RICH prefetching schemes. As expected, larger cache size improves the performance of all caching schemes, even if for $\hat{C} > 0.2$ the cache hit probability for RICH and netPredict becomes constant since the two policies do not fill the cache. However, RICH outperforms netPredict and POP up to 33% and 190% respectively, for small cache size. This is due to the higher effectiveness of the RICH policy, which tends to store chunks only in those ENs from where they can be downloaded with high probability, taking into account the available bandwidth and the dwell time distribution. For very large cache size ($\hat{C} > 0.75$), POP achieves higher cache hit probability as compared to RICH. This is because the cache is always fully occupied for POP and at $\hat{C} = 1$ each cache stores every content in the catalog, hence the hit is always guaranteed. Fig. 2.18 shows the network cache occupancy for the three caching schemes. By construction, regardless of the cache size, the cache occupancy is the highest for POP. In comparison with netPredict, the cache occupancy of RICH is slightly higher but the gain in the other performance metrics (e.g., cache hit probability, cache throughput and backhaul traffic) is significantly larger. Finally, Fig. 2.19 compares the cache throughput. RICH achieves up to 83 Mbps (i.e., around 11.9 Mbps per single EN/cache). In comparison, netPredict achieves up to 64 Mbps.

Backhaul network performance. The performance gain provided by RICH over both netPredict and POP can be observed also in terms of backhaul traffic, since the higher cache hit probability of RICH implies a lower probability to access the server and retrieve the content from there. Fig. 2.20 shows the overall backhaul traffic due to cache misses. In the best case, RICH reduces the backhaul traffic by approximately 57% and 70% as compared to netPredict and POP, respectively. Clearly, such reduction leads to a lower backhaul congestion.

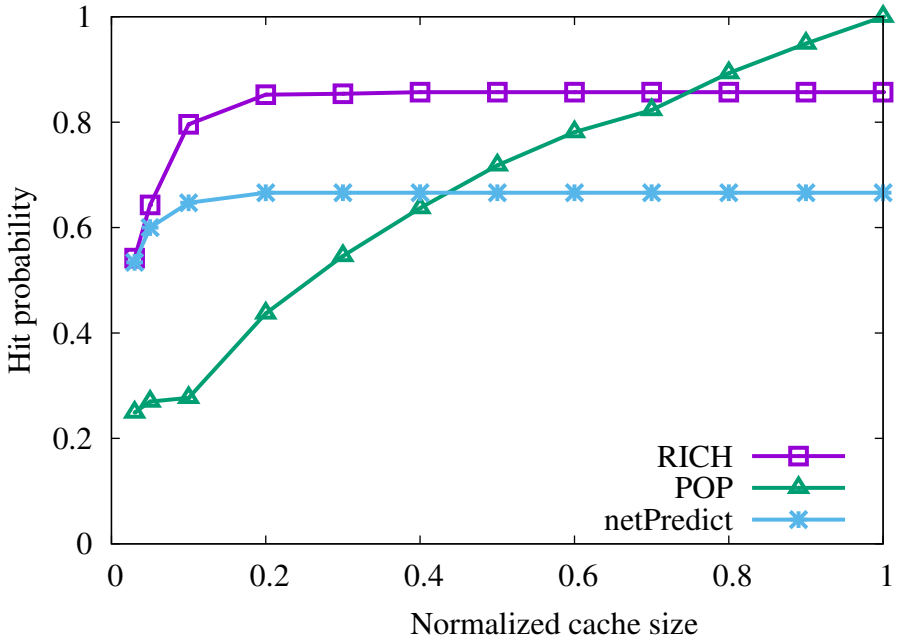


Fig. 2.17 Cache hit probability for the urban Bologna scenario.

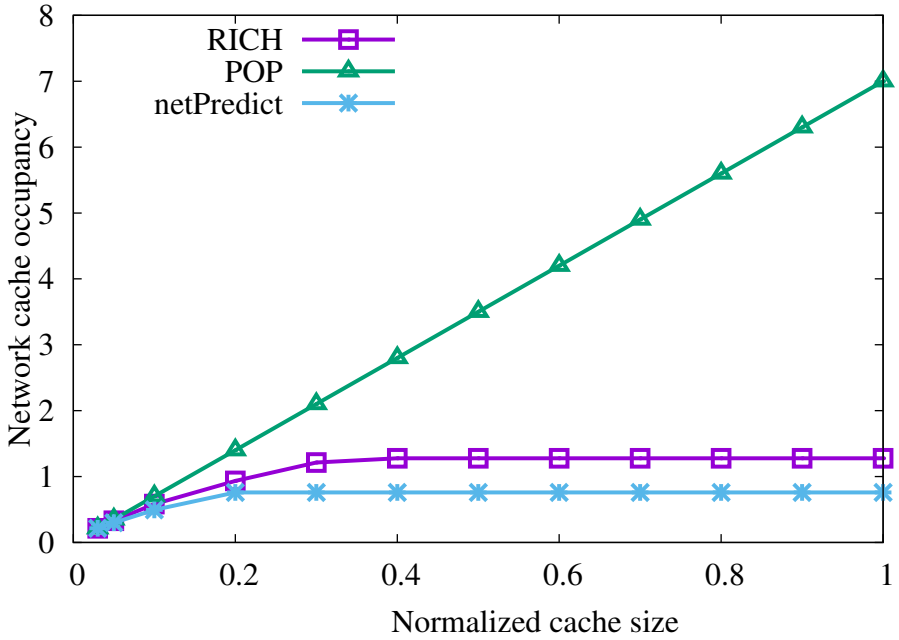


Fig. 2.18 Cache occupancy for the urban Bologna scenario.

Fig. 2.21 shows that RICH incurs lower backhaul overhead as compared with the other policies for enough large caches. In the best case, RICH achieves 27% and 67%

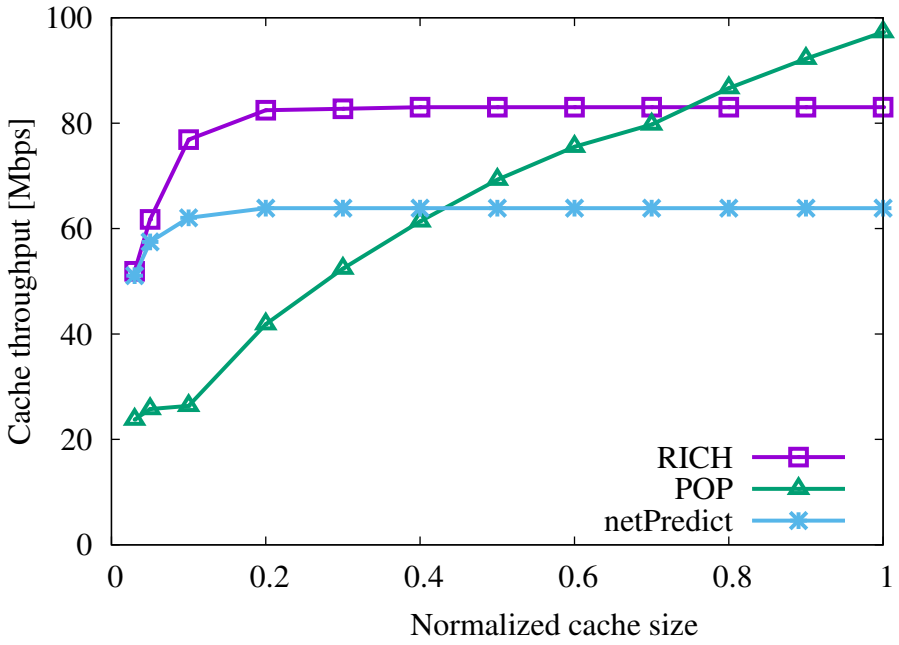


Fig. 2.19 Cache throughput for the urban Bologna scenario.

lower overhead than netPredict and POP, respectively. The lower negative values of the backhaul overhead indicate better reuse of the chunks stored in the cache. For small caches, RICH incurs higher backhaul overhead due to higher number of prefetched chunks and lower chunk reuse, but RICH still achieves lower backhaul traffic as compared with the two policies as previously shown in Fig. 2.20 thanks to the higher hit probability and content reuse.

A joint user/operator view. In order to understand better the multi-objective optimization problem we address, we also compare the performance of the caching schemes in terms of joint user/operator utility functions. The user utility is described as a measure of the advantage for the user. We model it as an exponential function decreasing with $1 - P_{hit}$, as shown in Fig. 2.22, since larger P_{hit} implies smaller latency to access the contents. The operator utility, which is the measure of the advantage for the operator, depends mainly on cache size, since this is a limited resource. We model it as an exponential function decreasing with the normalized cache size, as shown in Fig. 2.23. Following a standard approach, we formulate the joint utility function as product of the user and the operator utilities, and the result is shown in Fig. 2.24. It can be seen that the joint utility of RICH is better than the one of netPredict, regardless of the cache size. As compared to POP, the joint

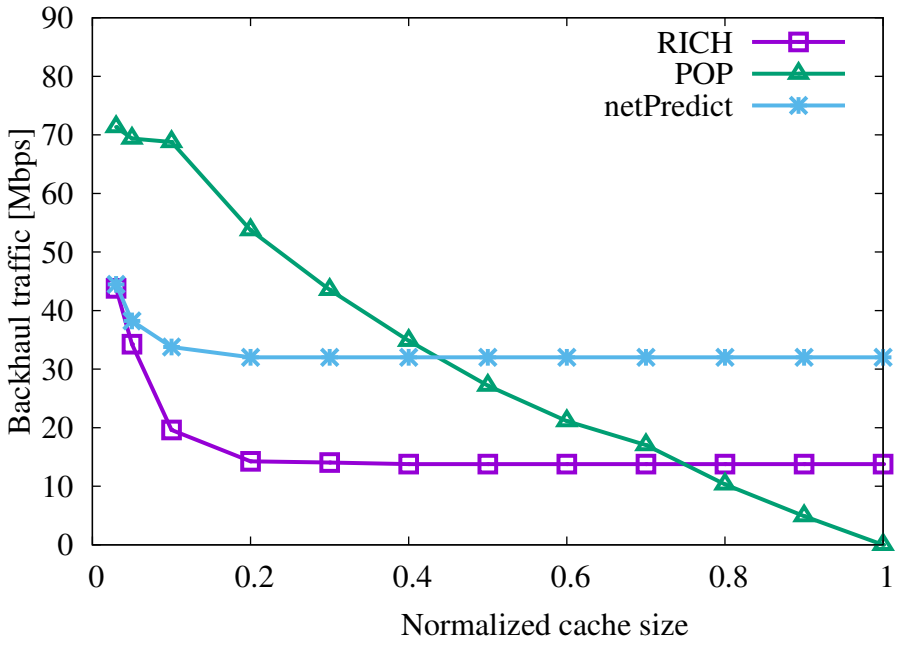


Fig. 2.20 Backhaul traffic for the urban Bologna scenario.

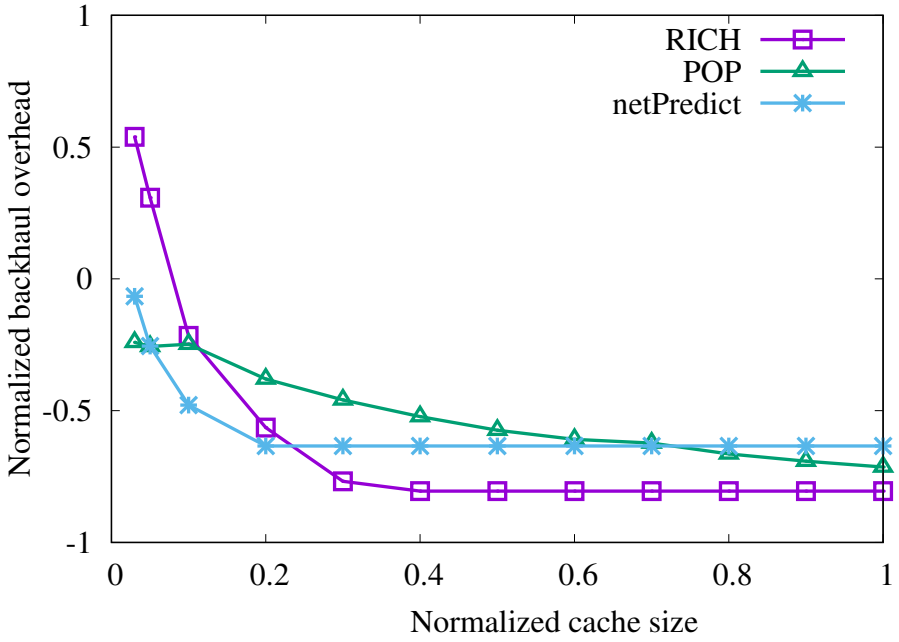


Fig. 2.21 Normalized backhaul overhead for the urban Bologna scenario.

utility of RICH shows significant improvement at small cache sizes. Whereas, it is

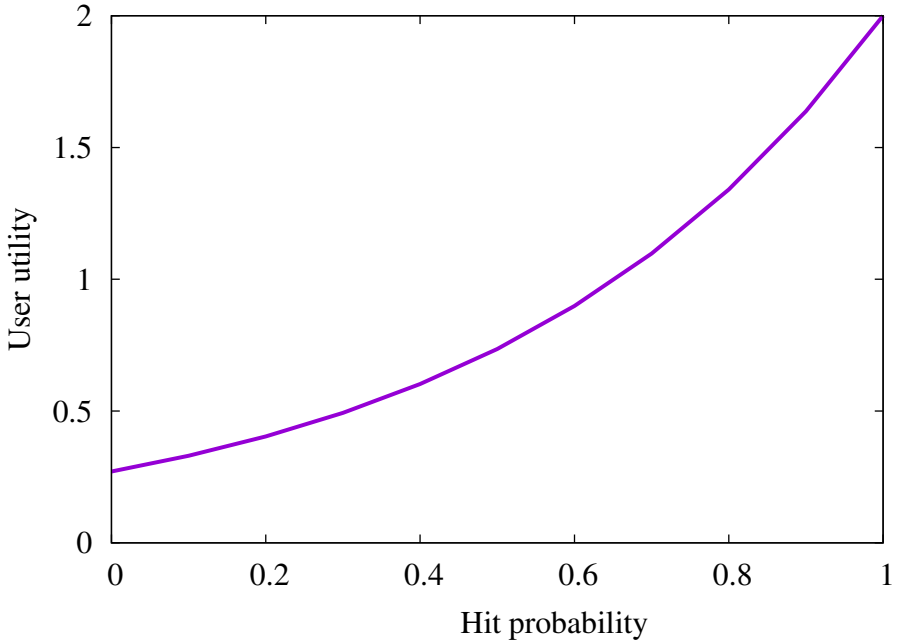


Fig. 2.22 User utility as a function of cache hit probability.

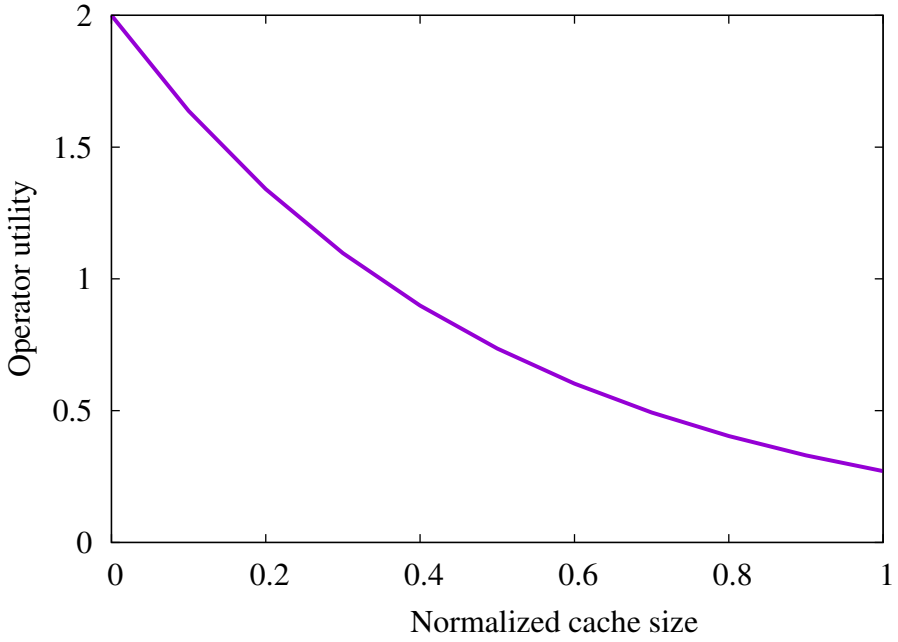


Fig. 2.23 Operator utility as a function of normalized cache size.

comparable to that of RICH scheme at larger values of the normalized cache size but at the cost of much higher cache occupancy.

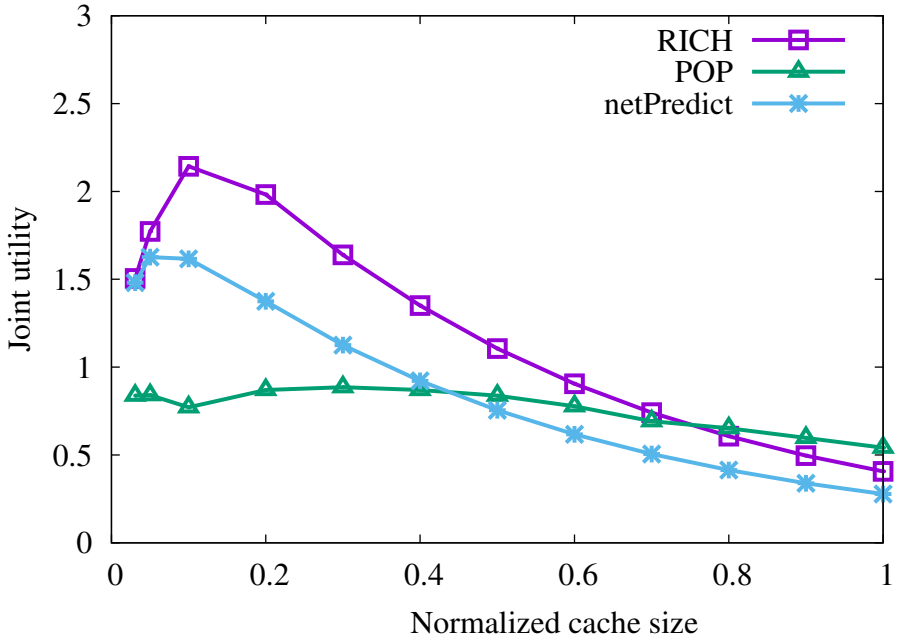


Fig. 2.24 Joint user/operator utility function for the urban Bologna scenario.

2.5.1 Errors in the knowledge of car mobility

To assess the robustness of RICH, we evaluate two kinds of error in the knowledge of the car mobility. The first error affects the dwell time under an EN, and thus the actual number of downloaded chunks. The second error affects instead the knowledge of the car path in terms of sequence of traversed ENs. As a reminder, RICH does not require any information about the detailed trajectory of the car.

Random shift in dwell time. In order to understand the robustness of our approach, we add a random error ε for the dwell time experienced by a car. Let W be the observed dwell time of a car under a given EN, and let $w_{\min}$ be the minimum observed dwell time under the same EN, based on past statistics. We set the actual dwell time W' of the car as:

$$W' = \max\{w_{\min}, W + \varepsilon\}$$

where ε is Gaussian distributed with average μ and standard deviation σ. When $\sigma = 0$ s, all the dwell times under an EN are deterministically shifted by μ. When $\varepsilon > 0$, the car is slowed down and when $\varepsilon < 0$ the car accelerates, with respect to the original speed.

Fig. 2.25 shows the cache throughput for different values of μ and for $\hat{C} = 0.4$ in the urban Bologna scenario, where the average dwell time across all the ENs is 39 s. For $\mu < -50$ s, most of the cars spend the minimum time under the coverage of the ENs ($W' \approx w_{\min}$), thus the cache throughput becomes very low. The cache throughput increases with μ as the cars get more time to download the chunks available in the cache. But, for $\mu > 20$ s, the high coverage time of the cars does not have a significant effect on the cache hit events as all the stored chunks have already been downloaded by the users. So, the cache throughput decreases due to the cache miss events. Overall, the cache throughput for fixed shift ($\sigma = 0$ s) is quite similar to that of random shift with $\sigma = 10$ s, which shows the robustness of our policy. However, the curve for $\sigma = 60$ s appears more flat due to the averaging effect of larger variance in dwell time, and the maximum throughput decreases by 13%.

Fig. 2.26 shows the cache hit probability at different values of μ. For $\mu < 0$ s, the cars spend less time under coverage of the ENs, thus both caches hits and misses decrease. However, the cache hit probability remains high due to certain presence of a car under the first EN. For $\mu > 0$ s, the cache hit probability decreases rapidly by increasing μ. This is because of the much higher number of cache miss events when the cars spend unexpected long time under the coverage of the ENs. The hit probability for $\sigma = 0$ and 10 s are similar. In the worst case, for $\sigma = 60$ s, the cache hit probability decreases at most by 30%.

Figs. 2.27 and 2.28 show the backhaul traffic and the normalized backhaul overhead. Both metrics are low when $\mu < 0$ s, as the cars spend less time under coverage of the ENs, hence the number of cache miss events is small and few requests reach the Data Store. Instead, when $\mu > 0$ s, both metrics increase due to the significant increase in the number of cache miss events. Notably, due to the large content reuse, the backhaul overhead is negative.

As a summary, only large errors in the dwell time (i.e., large $|\mu|$ or large σ) are affecting the performance, since the past statistics about the dwell time are largely "useless". Instead, if the past statistics are quite correct (small μ, σ) the performance are only slightly affected, confirming the robustness of the proposed approach in terms of errors in the dwell time.

Error in car path. In the considered scenario, RICH relies on the information of just two subsequent ENs traversed by a car along its path and it runs only when the car is under the first EN, thus only some uncertainty on the second EN is present;

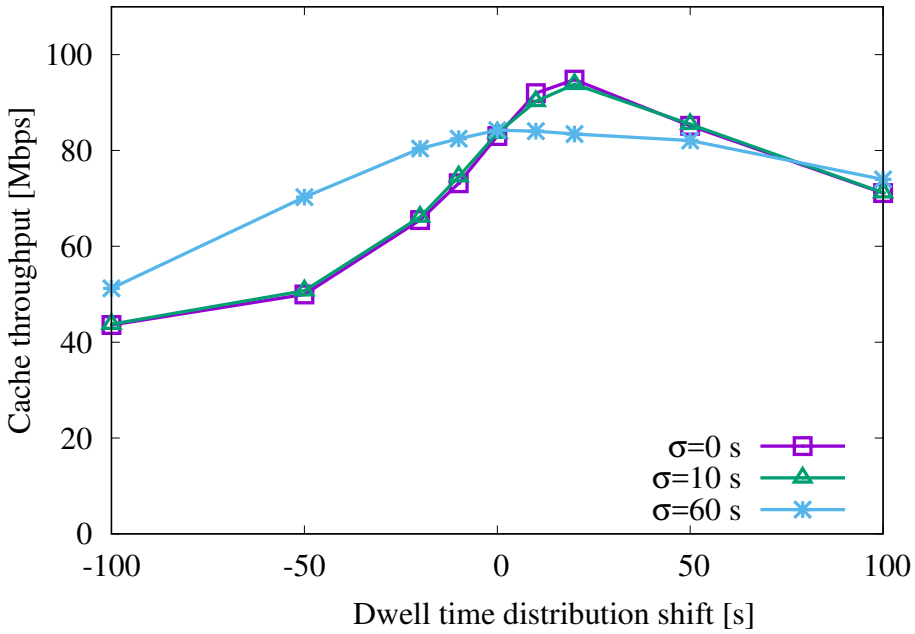


Fig. 2.25 Effect of random error ε in dwell time on the cache throughput for the urban Bologna scenario.

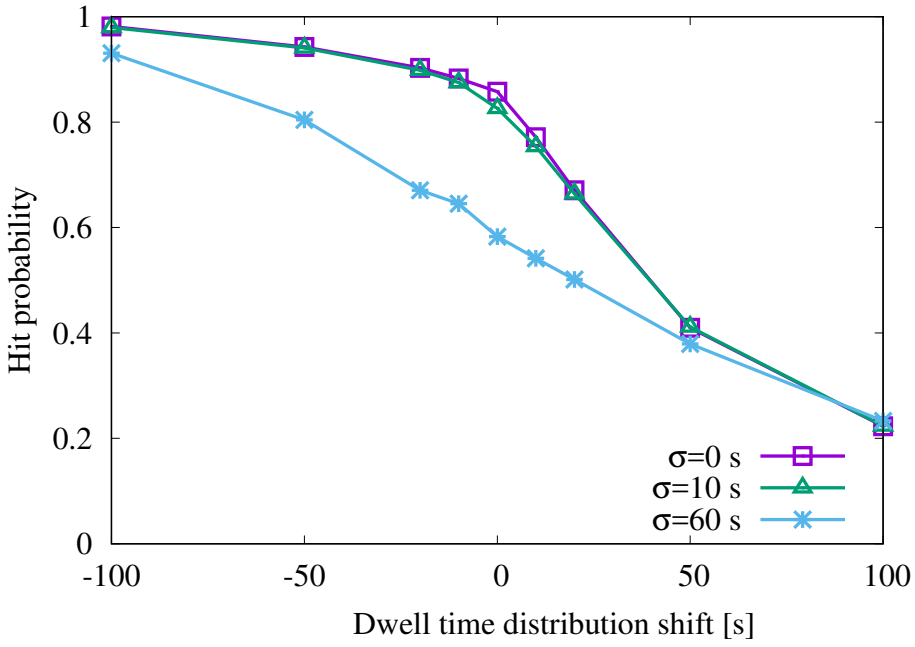


Fig. 2.26 Effect of random error ε in dwell time on the cache hit probability for the urban Bologna scenario.

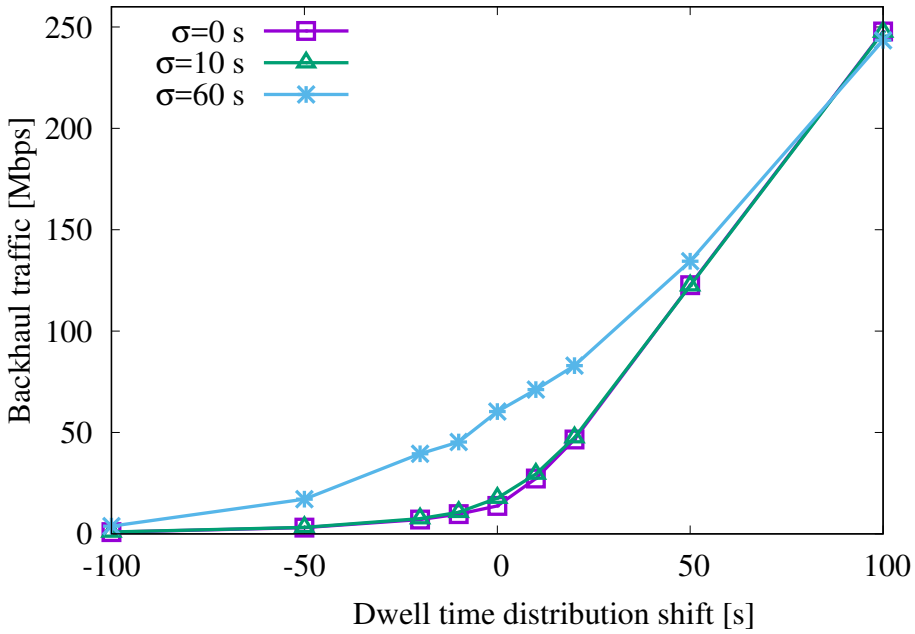


Fig. 2.27 Effect of random error ε in dwell time on the backhaul traffic for the urban Bologna scenario.

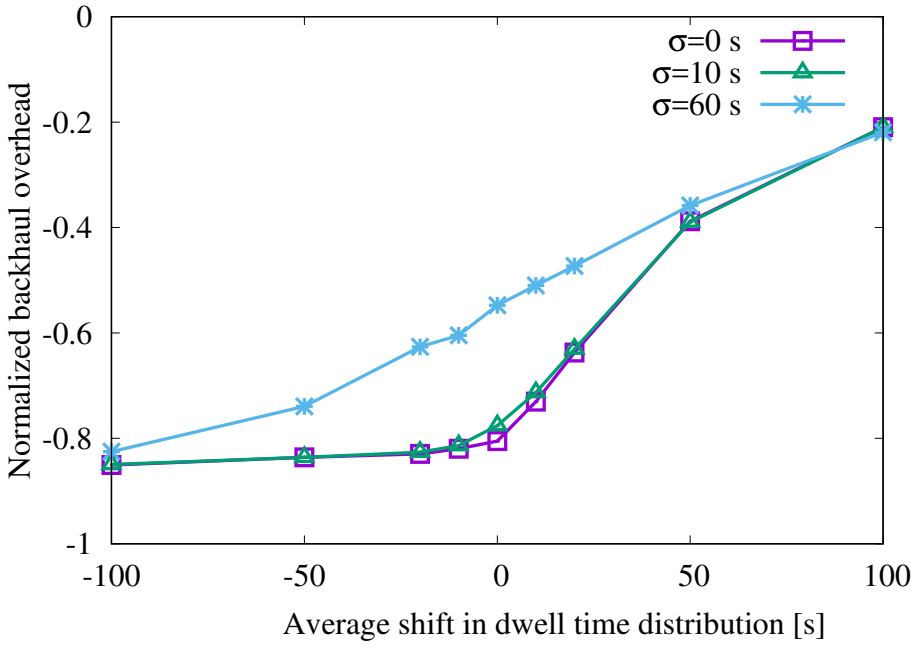


Fig. 2.28 Effect of random error ε in dwell time on the normalized backhaul overhead for the urban Bologna scenario.

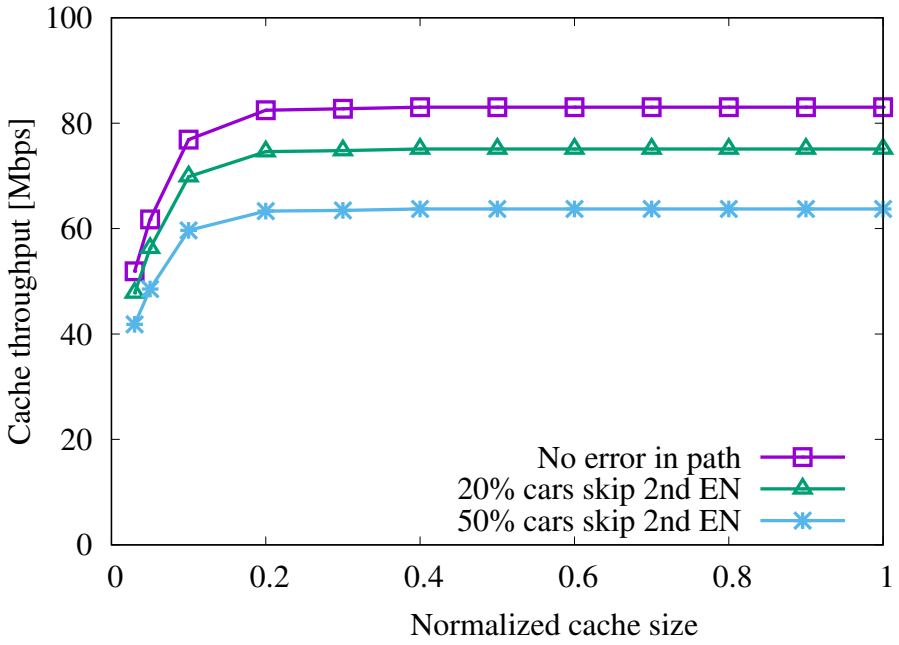


Fig. 2.29 Effect of error in car path on the cache throughput for the urban Bologna scenario.

this guarantees some intrinsic level of robustness. Nevertheless, the car could change its actual trajectory and completely skip the second EN along its expected path. So, the actual cache hit probability could arbitrarily change due a change of trajectory. For example, if all the cars that originally experienced a cache hit event under an EN according to the expected path would skip such EN from their trajectory, the corresponding cache hit probability would go to zero. Similarly, the cache hit probability would be 1 if only the cars that experienced a cache hit event in the original path would be considered. Thus, "adversary" errors in the second EN in a path can arbitrarily affect the cache hit probability (at least in half of the ENs), but the overall effect is meaningful only when all the paths change, i.e., the a-priori path knowledge is wrong. We do not expect this being a realistic case in our scenario.

To understand the effect of non-adversary errors, we selected at random a set of 20% or 50% of cars that would skip the second EN along their path, in the urban Bologna scenario. Fig. 2.29 shows the cache throughput (averaged over several simulation runs) against the cache size. The cache throughput decreases by approximately 10% and 23% with respect to the case with the original path, confirming the robustness of the proposed approach.

2.5.2 A more detailed knowledge on car mobility

In the above results, the Prefetcher relies on a rough estimate of car mobility, since it knows only the sequence of traversed ENs and the empirical distribution of the dwell time under each EN. We now analyze the beneficial effect of more detailed information about the car mobility. In this regard, we classify the cars in two categories: fast and slow. Slow cars are defined as the ones stopping inside the coverage area of a EN for more than 10 seconds; all the other cars are classified as fast. Table 2.7 shows the coefficient of variation of the resulting dwell time $\widehat{X}_i$ with and without such classification. As expected, for all ENs the variance of the dwell time for slow and fast cars are smaller, thus showing the gain of information when considering the proposed classification.

Figs. 2.30 and 2.31 depict the cumulative distribution corresponding to $\hat{X}_A$ for EN A and $\hat{X}_H$ for EN H in the Bologna urban area, for cache size equal to 2600 chunks. For the EN A, Fig. 2.30 shows that the users in slow and fast cars can download, on average, around 100 chunks and 30 chunks, respectively. The difference between these values is clearly due to the different average car speed. Observing the average number of cars in Table 2.2, EN A is expected to be located at a very congested intersection. Indeed, only few, slow cars experience a large enough dwell time and a small radio congestion that allow to download all the 2600 chunks stored in the cache. EN H, instead, appears to be located at a much less congested intersection, according to Table 2.2. This implies a lower radio congestion, hence, a higher download capability, as shown in Fig. 2.31. Indeed, under EN H, fast users download at least 170 chunks, with an average around 250, whereas slow users download around 1000 chunks. For the sake of brevity, we omit the distribution of the number of chunks downloaded from the other ENs. The number of fast and slow cars under each EN is reported in Table 2.6.

We re-ran the simulations based on the dwell time distributions conditioned on the fast/slow cars and we were able to further enhance the performance of the RICH and the netPredict approaches. Instead, POP policy, being mobility agnostic, is not affected with the classification of the users. Fig. 2.32 shows the cache throughput achieved by all the three caching schemes. Thanks to a better mobility knowledge, the cache throughput of RICH reaches approximately 91 Mbps, while netPredict achieves 83 Mbps. Similarly the backhaul traffic is further reduced to 6 Mbps and 13 Mbps for RICH and netPredict, respectively.

Table 2.6 Number of fast and slow cars at each EN

EN A		EN B		EN C		EN D		EN E		EN F		EN G		EN H	
Fast	Slow	Fast	Slow	Fast	Slow	Fast	Slow	Fast	Slow	Fast	Slow	Fast	Slow	Fast	Slow
376	1204	948	562	182	345	215	167	0	337	1456	0	68	93	120	86

Table 2.7 Coefficient of variation of $\widehat{X}_i$ at each EN i.

EN	COEFFICIENT OF VARIATION		
	Fast	**Slow**	**Combined**
A	0.36	0.61	0.75
B	0.31	0.45	0.80
C	0.43	0.32	0.59
D	0.30	0.40	0.72
E	–	0.32	0.32
F	0.04	–	0.04
G	0.38	0.17	0.44
H	0.33	0.40	0.70

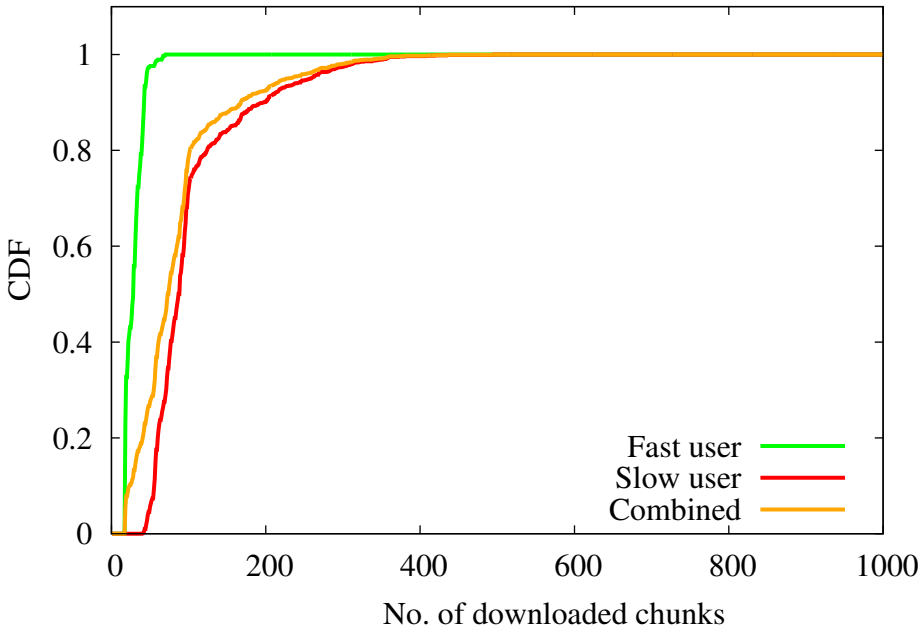


Fig. 2.30 Cumulative density function of the number of downloaded chunks $\widehat{X}_A$ measured in the trace, based on the slow/fast classification at EN A.

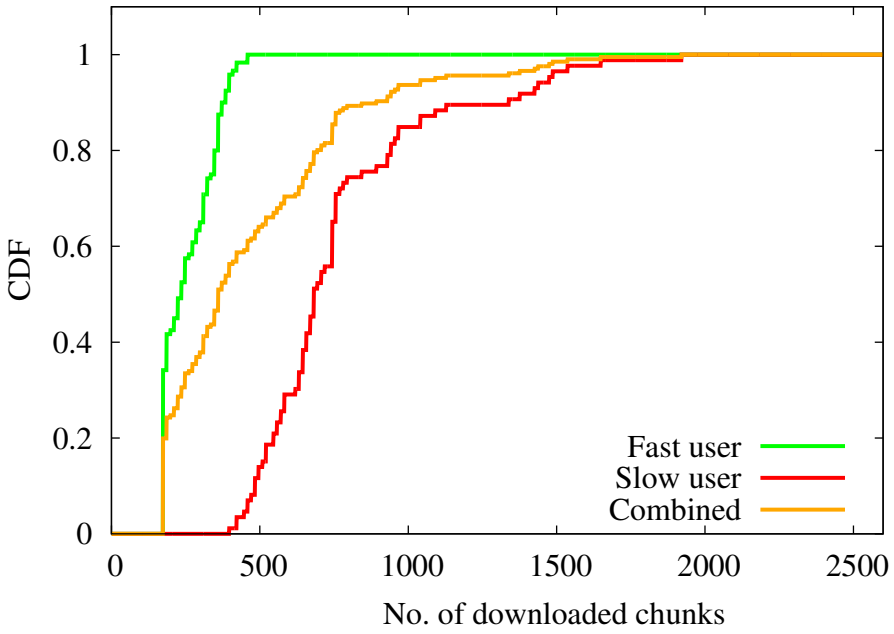


Fig. 2.31 Cumulative density function of the number of downloaded chunks $\widehat{X}_H$ measured in the trace, based on the slow/fast classification at EN H.

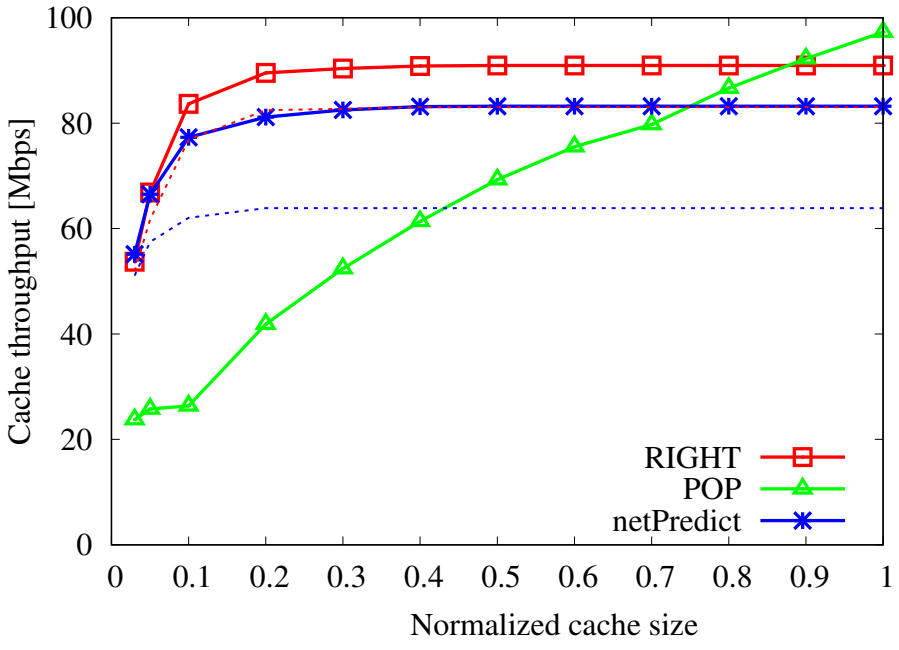


Fig. 2.32 Cache throughput, based on the slow/fast classification; the dotted lines show the performance without the fast/slow classification.

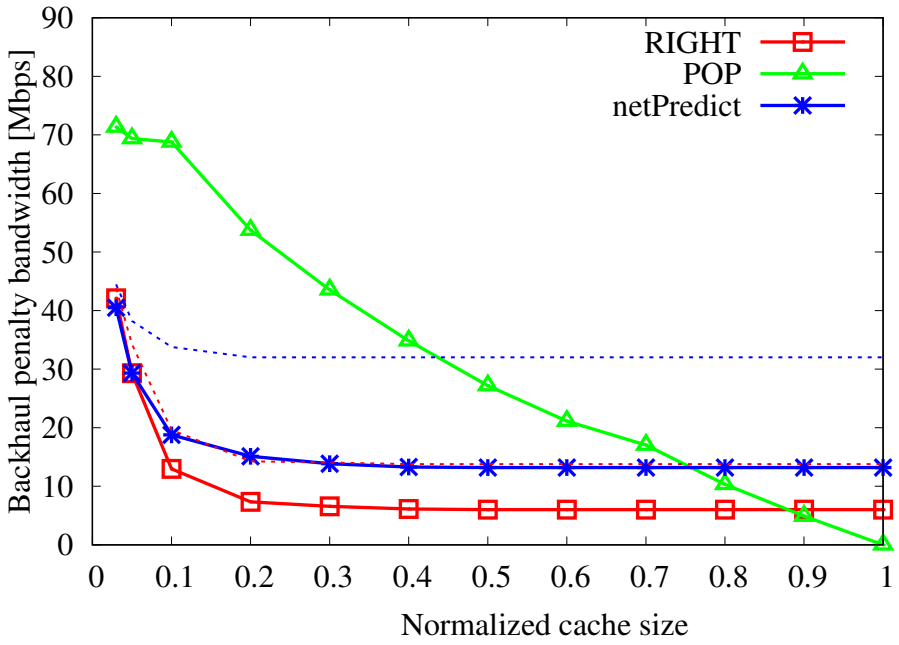


Fig. 2.33 Backhaul traffic, based on the slow/fast classification; the dotted lines show the performance without the fast/slow classification.

2.6 Related work

In the context of cellular networks, [16] devises an optimal geographic caching assuming that a user is covered by multiple base stations; this case is different from our work, since we do not consider overlapping coverage areas among ENs. Furthermore, [16] proposes an optimal probabilistic content placement policy that maximizes the total cache hit probability for random network topologies, based on content popularity. Thus, the policy there is oblivious of the actual mobility pattern of users, differently from our RICH policy, which leverage some coarse information about the user mobility. In a hybrid scenario comprising MANET and cellular networks, [17] proposes an optimal caching and routing policies. Each node estimates locally the content popularity and stores the content in the cache based on its popularity. This scheme can be considered as a distributed implementation of the POP policy that we use for comparison in our work.

For the specific case of cellular backhaul networks, [26] investigates the effect of different criteria to identify the web content adopted when accessing the caching system. The main idea is to avoid duplicated content items in the caches, since

those same items could appear with different identifiers at application level. In our scenario, instead, we specifically consider the streaming of content through a chunk-based approach, for which we assume that each chunk and each content are univocally identified.

Few works have employed cooperation among caching nodes to improve cache performance in heterogeneous cellular networks. The caching architecture proposed in [27] is hierarchical and assumes caches in both small-cell and macro-cell base stations, and a group of small-cell base stations constitute a cache ensemble and cooperate together to serve the users under coverage. The user content request can be satisfied either from the group of small-cell base stations or the macro-cell base station. Similarly, the work in [28] considers a cluster of small-cell base stations as one cache entity. The base stations adopt both cooperative caching techniques and cooperative transmissions. Our approach is instead based on a centralized prefetching scheme orchestrating the caches.

We remark that all the above cited caching schemes are oblivious of user mobility. Instead, we assume to know the sequence of waypoints of the temporal and spatial trajectory followed by the cars. This information can be deduced from car navigation systems or it can be easily predicted. For example, studies such as [29] and [30] suggest that people usually drive on familiar routes (drive to work, school, etc.) and this can be exploited to develop quite accurate prediction models. On this regard, [31] proposes a mobility prediction scheme, based on the previous history of users, which improves content distribution in vehicular networks through simpler handover procedures.

Few works have investigated caching schemes specifically taking into account user mobility. The work in [14] proposes an approach, called netPredict, to address a scenario very similar to ours, based on an architecture denoted as “MobilityFirst", introduced in [32], which aims to support smooth mobile content delivery when users move across the network. In MobilityFirst, a global identifier is associated to each user, so the user mobility is recorded at each node. In terms of caching, each node is equipped with two distinct buffers. The first one caches the most popular content items, exactly as the POP policy considered in our work. The second buffer is instead devoted to store the content based on a prefetching policy leveraging the predicted sequence of nodes traversed by each particular user and the past information of dwell time under each node. In particular, netPredict prefetches the range of chunks by

considering the average dwell time and the average available bandwidth. A similar prefetching policy is proposed by [15] in a cellular network scenario. Similarly to our work, the content is delivered to users by base stations using a chunk-based approach. The specific mobility of each user is considered in order to identify the chunks to prefetch in the caches along the user path. Unlike our work, however, both [14] and [15] assume that the caching policy knows or predicts the spatial and temporal trajectory of each user, in order to estimate the time intervals in which the user will be covered by each base station. Our approach instead requires to know just the distribution of the dwell times under each edge node at aggregate level. This distribution can be estimated locally by each EN and does not require at all the precise knowledge of the car trajectory: only the sequence of ENs is needed. This simplifies the prediction process and preserves at some level the privacy of users.

Recently, efforts have been made to introduce Information-Centric Networking (ICN) paradigm within the context of vehicular networks. [33] proposes an architecture for content-centric vehicular networks, which is compatible with our proposed prefetching approach. Considering the ICN architecture, [34] uses computing, caching and communication resources of the vehicles for content storing and sharing; this is different from our work, since we assume that content is stored only in the ENs and not in the vehicles. In standard ICN based architectures, network nodes reactively store contents; the data is cached during delivery to the user. However, similarly to our approach, [35] and [36] propose to prefetch contents in ICN nodes. In detail, [35] considers a simplified scenario of a car moving along a motorway covered by roadside units with overlapping coverage areas. The proposed prefetching approach is based on the local popularity of the content (evaluated through the frequency of ICN "interest" packets) and on the speed of the car entering inside the coverage area of multiple roadside units. Assuming constant speed for the whole trajectory under coverage, the central module computes the sequence of chunks that will be downloaded at each roadside unit and then uploads the chunks in advance. This approach cannot be applied in our urban scenario where the velocity is highly variable, since affected by random traffic conditions, which is instead considered in our RICH policy. The study in [36] formulates the problem of optimally placing content chunks in the ICN-based network nodes as an integer linear programming optimization problem. The main objective is to maximize the retrieval probability of a content file for a user that moves around the network. Moreover, forward error correction coding is adopted to reconstruct the whole content if enough chunks have

been received, independently from their order. This work is thus based on a single content retrieval and not on content streaming as in RICH case, where in-order delivery is required.

2.7 Summary

In this chapter we study how to efficiently provide connected cars with streaming data as they drive along a road covered by wireless Edge Nodes (ENs). Our RICH prefetching policy determines the content chunks to store in the edge node caches, based on the past statistics of the achievable data rates and of the dwell time experienced by the cars under the coverage of each edge node. RICH requires only to know the sequence of ENs traversed by a car, without any detailed information about its actual trajectory and real-time traffic conditions. As future work, we intend to extend our work in order to address the case when the sequence of ENs are only statistically known.

Throughout extensive trace-driven simulations, our scheme was shown to improve the cache throughput and to reduce the backhaul traffic, with beneficial effects both for the users and the network operators. Furthermore, RICH was shown to be robust to possible errors in the knowledge of the path and of the dwell time.

Chapter 3

The Cache Node Control in NDN

3.1 Motivation

An increasing usage of content-based applications such as video sharing, social media networking, and e-commerce, has led to a dominant use of the Internet as a *content distribution network* (CDN). However, implementing content distribution in legacy IP networks is challenging. Indeed, the communication model in legacy IP networks is based on the packet exchange between pairs of hosts, which requires mapping content to location endpoints as well as deploying CDNs or P2P networks [37]. In this scenario, Named Data Networking (NDN) emerges as a promising paradigm in which hosts address the content in network packets rather than contacting the host containing the content, hence greatly simplifying content distribution. Moreover, each network node in NDN is equipped with a content store, which caches a copy of already delivered contents. In this way, future content requests can be satisfied from the network nodes instead of the content server.

Content caching is therefore a key component of NDN, and significantly benefits both users and network operators. From the user's perspective, the ability to retrieve content from intermediate nodes in the network, reduces delays and enhances the quality of experience. From the operator's point of view, the network overhead is greatly reduced, especially if multiple users request the same content (e.g., popular videos and live sport streams). In light of such advantages, future 5G systems will integrate caching capabilities in the network, especially to provide media and entertainment services [3].

Implementing NDN requires, however, a drastic change in the legacy network infrastructure. In this regard, Software Defined Networking (SDN), which also represents one of the core technologies in the network evolution towards 5G [4], can be leveraged to realize the NDN concept. Indeed, in SDN control plane and data plane are separated, and a logically centralized controller programs the data forwarding plane by keeping a global view of the network. As a consequence, SDN enables a programmable network, lowers operational costs and allows for flexible services. This allows us to easily install the functionality of an NDN node on top of the SDN controller that is responsible to process and react to NDN packets received from all the switches in the data plane.

An existing implementation of the SDN paradigm is OpenFlow (OF) [38], which is a standard protocol enabling the communication between the SDN controller and the network devices. In OF, the controller installs match-action forwarding rules on the network switches. Since the switches are stateless and the rules are static, the switch must interact with the controller for any "new" action or change of action to be performed in the data plane, hence leading to a large communication delay and a large amount of control traffic. In many time-critical network applications, where contacting the controller may be unfeasible, some control logic can be delegated to the switches, which allows modification of the forwarding rules based on local network events. Such an approach, enabling swift local decisions at the switch without involving the controller, requires to support states within the switches and is called *stateful* data plane, in contrast to the vanilla stateless OF data plane. OpenState [7] is a recent extension of OF that enables a stateful SDN approach, leveraging a standard OF switching architecture.

In this work, we combine the stateful SDN approach with the NDN technology, and realize the functionality of the NDN node by attaching a local cache to a stateful SDN switch, implemented through OpenState. The switch autonomously provides the required content provisioning functionality without interacting with the SDN controller. The main advantages of our approach are the simplicity, since the NDN control logic is embedded directly within the switch, and the smaller latency with respect to vanilla SDN, since the NDN control logic does not need any interaction with the controller to operate. In particular, our main contributions are as follows:

- *Architecture:* we propose an NDN network architecture that enables direct support of caching within SDN switches;

- *S/N-DN node:* we design a stateful SDN/NDN (S/N-DN) node, i.e., a stateful SDN switch capable to perform the functionality of an NDN node;
- *Implementation:* we provide a stateful data plane implementation of the S/N-DN node using OpenState, thus avoiding any interaction with the controller at runtime;
- *Performance evaluation:* we evaluate the performance of our solution in an emulated environment considering several relevant metrics. Moreover, we benchmark our solution with a traditional, stateless OF data plane implementation of the S/N-DN node.

The remainder of the chapter is organized as follows. Sec. 3.2 gives an overview of stateful SDN and of NDN, which are the two pillars of this part of the thesis. Sec. 3.3 introduces our solution, and Sec. 3.4 presents the data plane implementation of our approach. Sec. 3.5 describes the methodology we use to evaluate our solution, while Sec. 3.6 presents experimental results. Sec. 3.8 discusses related works and highlights the novelty of our solution. Finally, conclusions are drawn in Sec. 3.9.

3.2 Preliminaries

In this section, we give an overview of stateful SDN and NDN technologies as these are the two pillars on which our solution is based. First, we describe stateful SDN as well as the platforms that help to implement a stateful data plane. Then we explain the NDN architecture along with its main components.

3.2.1 Stateful SDN

Existing SDN technologies such as OpenFlow [38] forces separation of control and data plane of a network. As a consequence, all the network intelligence is contained in the logically centralized controller that is responsible to govern the "dumb" switches. This means that the controller becomes responsible for all the network-wide and local decisions, hence resulting in large signaling overhead and latency.

In stateful SDN, part of the network intelligence is moved from the controller to the switches; specifically, the controller is involved in making network-wide decisions since it has a global view of the network, while switches make decisions that only rely on switch-local states.

Stateful switches maintain states for all the incoming traffic flows, with each flow being identified by a flow key. Based on the current state, the switches apply match-action rules on the packets belonging to the flows, hence reducing the need to rely on the controller to make local decisions. In this regard, OpenState [7] extends OpenFlow to configure stateful data plane. In order to enable the stateful functionality, OpenState switches run programmable eXtensible Finite State Machines (XFSMs), where an XFSM implements two tables: a *state table* and a *flow table*. The state table stores the current state for each active flow, while the flow table stores the match-action rules. Differently from a vanilla OF flow table, in OpenState, one of the fields used for matching a flow is the current state and the action allows to change the state. Interestingly, this approach enables the implementation of the XFSMs using almost the same hardware architecture as vanilla OF switches, exploiting the efficient lookup in a Ternary Content-Addressable Memory (TCAM). Indeed, for any incoming packet, first its state is looked up from the state table, then the action and the next state are chosen based on both the current state and the packet header, according to the flow table. When accessing the state table to lookup and update the state of a flow, OpenState defines two different keys: the *lookup_scope* defines the packet header fields used for lookup, and the *update_scope* defines the ones used for update. Their distinct definitions allow a more flexible packet processing, as discussed in [7]. The aforementioned packet flow inside a stateful stage/XFSM is also shown in Fig. 3.1. The complete OpenState protocol specification is available at [39], while the feasibility of implementing hardware-based OpenState switches is addressed in [40].

P4 [41] is an alternative approach to OpenState to implement stateful SDN. Specifically, P4 is a high level programming language used to describe the packet processing within a switch with a much higher flexibility than OpenFlow, in which the set of available matching fields is fixed. Indeed, P4 allows an arbitrary definition of the packet parsing and processing, and thus enables programmable, protocol independent switches. A P4 program is developed obliviously from the underlying switching architecture and a P4 compiler has the responsibility to tailor the required

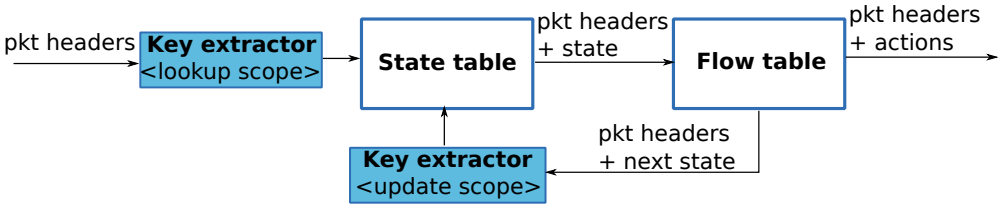


Fig. 3.1 Basic packet flow inside an XFSM/stateful stage in OpenState.

packet processing to the underlying hardware architecture, designed according to a specific processing model [42, 43].

Unlike OpenState, where the states are embedded within the available TCAMs, in P4 the states are available through external objects, thus leaving stateful data plane implementation as optional. For this reason, we have implemented the NDN node through OpenState, which natively supports a stateful data plane in OF-compatible switches.

3.2.2 Named Data Networking (NDN)

Named Data Networking (NDN) [5] is an architecture to support the Information Centric Networking (ICN) approach that aims to change the traditional IP network into a content-oriented one. In legacy IP networks, packets in the network are destined to specific IP addresses; in contrast, NDN packets are destined to a content, thus the routing is based on content identification (i.e., data names). Moreover, each network node in NDN is able to cache contents, thus content requests can be satisfied by multiple nodes in the network. The routing is anycast and based on two types of packets: Interest and Data, and both carry a content name. This name is hierarchically structured, similar to URLs, and each name is divided into components, e.g., `/polito/tng/courses/A.pdf`. The Interest packet is generated by the *data consumer* and sent to the NDN network. The network nodes forward the Interest packets towards the *data producer* (i.e., the server hosting the content) based on the content name. The Data packet gets back to the consumer by following the reverse path of the corresponding Interest packet. The nodes typically cache the contents received back from the data producer. Thus, in the case a node receives an

Interest packet for a locally stored content, the node will answer directly with the Data packet, avoiding the forwarding of the packets to/from the data producer.

In order to manage the forwarding of Interest and Data packets, each NDN node features the following data structures:

1. Content Store (CS), which caches a copy of already requested contents;
2. Cache Lookup Table (CLT), which allows to understand if a requested content is already stored in the CS;
3. Pending Interest Table (PIT), which stores forwarded Interests along with the interface from which the Interests were received, when they cannot be satisfied from the CS;
4. Forwarding Information Base (FIB), which contains forwarding rules based on the content names; it is the equivalent of IP routing tables in the context of legacy IP networks.

When an Interest packet arrives at an NDN node, the latter first extracts the content name and checks the availability of the requested content in its CS through the CLT; if the content is available, then the Data packet is sent back to the consumer. Otherwise, the content name is looked up in the PIT to find any matching entry; if it exists then the incoming interface of this Interest is inserted in the PIT. In case there is no matching entry in the PIT, then the Interest is forwarded towards the producer based on the FIB. At this point, a new entry is created in the PIT, which records the pending request in terms of content name and incoming interface. Inside the FIB, the node performs longest prefix matching on the content name to forward the Interest packet; if no matching entry exists in the FIB, then the Interest is discarded.

When a Data packet arrives, the content name in the packet is looked up in the PIT and the content is forwarded to all the interfaces listed in the PIT entry. Once the content is transmitted towards the consumers, the PIT entry is deleted. In addition, a copy of the content is cached in the CS and the CLT is updated, so that future Interest packets can be locally satisfied. In the case the CS is full, one content is evicted to make space for the new content to store, and the CLT is updated accordingly.

3.3 The Stateful S/N-DN Approach

Here we propose our stateful S/N-DN solution, which combines the SDN and NDN technologies, i.e., we equip a stateful SDN switch with the components that allow us to implement the NDN approach therein. As a result, we can make NDN-related decisions within the switch without interacting (at runtime) with the SDN controller.

As discussed in detail in Sec. 3.8, previous works have combined SDN and NDN, but failed to fully integrate the two technologies. Indeed, stateless SDN switches cannot make local decisions on how to process NDN Interest/Data packets, thus a dedicated agent, external to the switch, is typically envisioned for the actual packet processing. Our stateful SDN approach, instead, integrates the NDN data structures (CLT, PIT) directly within the switch and avoids the interaction with an external NDN agent.

We begin this section by explaining our proposed stateful S/N-DN architecture. Then, we highlight the challenges we face, along with the solutions we envision to enable NDN in an OF network. Finally, we present a toy example to better clarify the proposed solution.

3.3.1 The proposed architecture

Owing to the benefits of incorporating caching capabilities in an SDN network, we propose that each OF switch is attached with a local cache. We call this node comprising the OF switch and the cache, *cache-equipped switch*. We envision that the cache-equipped switch, upon receiving content requests from a user, should primarily try to satisfy the request by exploiting its own cache. If the content is not available therein, the request should be remotely satisfied either from the data producer or from a cache-equipped switch along the path towards the data producer. Although the above mentioned functionality can be provided by either following the stateful or the stateless approach, we envision that the cache-equipped switch functions in the stateful manner, enabled by OpenState technology. In this way, the switch can make decisions on its own, by maintaining the states of pending requests, thereby eliminating the need to interact with the controller at runtime. It follows that this approach requires no control traffic (at least to operate the NDN control logic) after bootstrapping the switches, which reduces the end-to-end user latency.

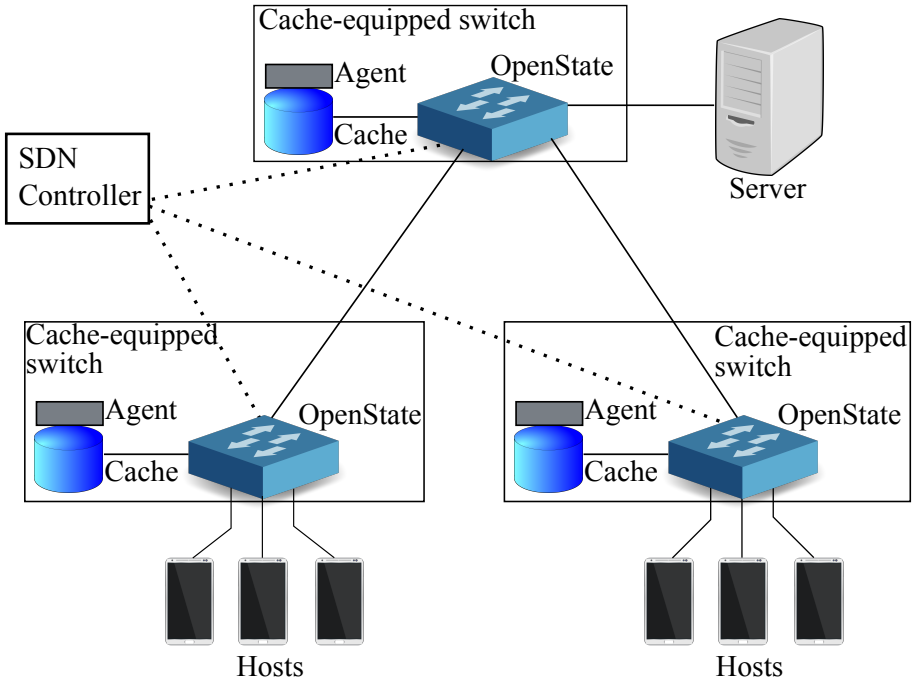


Fig. 3.2 The Stateful S/N-DN architecture.

Our network architecture, as shown in Fig. 3.2, is based, for simplicity, on a single SDN domain. The SDN domain consists of cache-equipped switches, an SDN controller and the hosts, which are typically the terminals handled by users. The cache-equipped switches are connected with a tree topology to the server, which acts as the data producer and owns the complete catalog of the requested contents. As mentioned earlier, the cache-equipped switch firstly tries to satisfy the content requests using its own cache, otherwise the request is forwarded towards the server. In response, the server sends the requested content to the host, and a copy of each content is stored in the switch cache for future requests. It is fair to assume that the delay for content retrieval is negligible when the content is stored in the cache within the switch.

Importantly, the cache-equipped switches must forward requests based on the content name, which is not possible in legacy IP networks. Each of the hosts in Fig. 3.2 acts as an NDN consumer that generates Interest packets. The server acts as an NDN producer, which responds to the hosts with Data packets. Finally, the functionality of an NDN node is implemented inside the cache-equipped switch.

Hence, we name the cache-equipped switch with NDN functionality as *S/N-DN Node* (i.e., an integrated SDN-NDN node) and use this term hereinafter. As explained in Sec. 3.2.2, the main components of an NDN node are: PIT, FIB, CLT and CS. Thanks to the availability of local states, the OpenState switch can support the behavior of the PIT and of the CLT, thus the switch can forward autonomously the content requests either to the cache or to the server.

The cache combines a *cache agent* and an actual storage, implemented with either volatile or non-volatile memory. The cache agent receives the messages from the switch and interacts with the local storage. The main functionalities are the following:

- whenever an Interest packet is received, the requested content is read from the storage and then a Data packet is sent to the requesting host; notably, thanks to the CLT implemented within the switch, an Interest packet is received only when the content is available in the local storage;
- whenever a Data packet is received, the content is written on the cache. In case of an eviction, a control message is sent to the switch with the fingerprint (as described in the following section) of the evicted content, to properly update the CLT implemented within the switch.

To support such interface functionalities between the switch and the cache, a simple embedded computer (e.g., Raspberry Pi) could be adopted.

In summary, all the functionalities of an NDN node can be implemented inside a stateful S/N-DN node. The behavior of the PIT, the CLT and the FIB can be supported by the OpenState switch whereas the local cache attached to the switch acts as the CS.

3.3.2 Challenges and solutions

It is not straightforward to enable NDN over an OF network. Notably, TCP cannot be adopted as transport protocol; indeed, it is not suitable for NDN because its end-to-end congestion and flow control are not suitable to interact with caching schemes and to support one-to-many communications, as required in an NDN architecture [44].[1]

[1] As one NDN Interest packet at most retrieves one Data packet, the flow control can be realized by controlling the sending rate of Interest packets. Moreover, congestion control in NDN can be

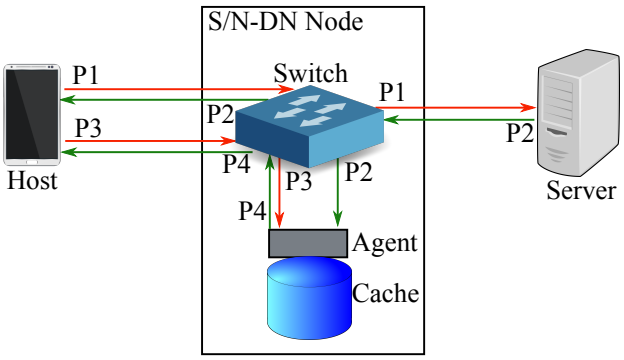
Thus, we use UDP/IP protocol, i.e., we consider that NDN Interest/Data packets are included in the payload of UDP packets. Second, an OF-switch cannot process NDN packets, as content identification is not possible. The OF protocol allows the switches to parse only a limited set of fields from an incoming packet, in particular it is not possible to parse the data field at the UDP layer. Following the approach introduced in [50–53], we piggyback the NDN semantics in the UDP/IP header fields. Specifically, we compute a fingerprint of the content name and store it in the UDP source and destination ports. Hence, the switch can identify the content referred by Interest packets using the fingerprint. Last, the available space in the packet header is limited; only 32 (16+16) bits are available in the UDP port fields. For variable name components, an option is to partition the available bits into groups, one for each name component. However, if the number of name components grows large, the number of bits per component decreases, hence the probability of fingerprint collision increases. The solution we recommend in such case is to store the fingerprint in other OF-matchable fields such as IPv6 source and destination addresses.

3.3.3 A toy example

In order to better explain our solution, we consider the toy scenario depicted in Fig. 3.3 (top), and use it to explain the main interactions between the network entities and the corresponding packet format, as shown in Fig. 3.3 (bottom). The scenario includes a host requesting content A, before and after the content is stored in the cache. For simplicity, we assume a simple Ethernet network connecting a host, a server and a cache to the OpenState switch. In the following, we report the sequence of exchanged packets. Notably, no interaction occurs between the switch and the SDN controller.

1. The host generates an Interest packet (P1) for content A and sends it to the switch inside the S/N-DN node. The hash of the content name, denoted by "hash(A)", is coded using 32 bits of the UDP source and destination ports, while the NDN Interest packet is inserted in the UDP payload. Since the content is not stored in the cache, the switch forwards the request (P1) to the server.

implemented by throttling the request rate and forwarding rate of Interest packets at the consumer and the NDN nodes respectively, hence, giving rise to receiver-based and hop-based congestion control mechanisms [45–49].



	Ethernet	IP	UDP ports	UDP payload
P1	Eth(Host)-->Eth(Server)	IP(Host)-->IP(Server)	hash(A)	Interest(A)
P2	Eth(Server)-->Eth(Multicast)	IP(Server)--> IP(Multicast)	hash(A)	Data(A)
P3	Eth(Host)-->Eth(Server)	IP(Host)-->IP(Server)	hash(A)	Interest(A)
P4	Eth(Cache)-->Eth(Host)	IP(Server)--> IP(Host)	hash(A)	Data(A)

Fig. 3.3 Exchange of messages in the toy example when content A is requested.

2. The server generates a response packet (P2) in which the Data is inserted into the UDP payload field. The response packet is sent to the switch inside the S/N-DN node, which does not only forward it to the host, but also places a copy of the packet in the local cache. In the response packet, the server uses multicast IP and MAC destination addresses since, in general, the data could be directed to multiple hosts, whose requests were pending in the switch.

3. The host generates another Interest packet (P3) for the content A and sends it to the switch. Since the content A is already stored in the cache, the switch forwards the request to the cache instead of the server.

4. The cache agent sends back the content to the host through a Data packet (P4). This packet is destined directly to the requesting host, since content retrieval from the cache is instantaneous and it is not possible to have pending requests for a content in the cache.

3.4 Stateful Caching Implementation

We now describe the stateful data plane implementation of an S/N-DN node. To this end, as a first step, in Sec. 3.4.1 we consider an OpenState switch and the associated controller. On top of the controller, we develop an application to properly pre-configure the OpenState switch so that packet forwarding can be performed based on content names. This simple application lays the foundation for developing a more complete application (detailed in Sec. 3.4.2) that allows the controller to configure the stateful switch to work as an S/N-DN node. Note that this enhanced application enables the switch to provide a prototypical support for the NDN data structures (e.g., PIT and CLT), thus integrating the main NDN control functionalities within the switch.

3.4.1 NDN Packet Forwarding using OpenState

We describe a basic network application that controls the switch destination port of NDN Interest and Data packets based on the content name, instead of the usual forwarding based on layer-2/3 headers. We assume just one host acting as a data consumer and one server acting as a data producer, connected as in Fig. 3.3.

Each content is identified by a fingerprint applied on its name. The state is associated to the content. Depending on the current state, the request is forwarded to a distinct port, i.e., either the server or the cache. This behavior is described by the state machine diagram in Fig. 3.4. Specifically, there are three possible states: default, pending and stored. Initially, the state of any content is set to “default”. When a request for a content in default state is received, the request is sent to the server and the content becomes “pending”. When the content is pending, all new requests for it will be dropped, since redundant. On the contrary, in the case of a data packet from the server, the content will be sent to the cache and to the host and the content will become “stored”. When the content is stored, any request for the content will be forwarded to the cache, and the content data from the cache will be forwarded to the host. If the content is evicted from the cache, the cache agent notifies the switch using a dedicated packet (e.g., adopting a special MAC address), which triggers the state of the content again in the “default” state. Table 3.1 shows the actual flow table that is installed on the switch by the controller, just during the bootstrapping of the S/N-DN node, and that implements specifically the XFSM

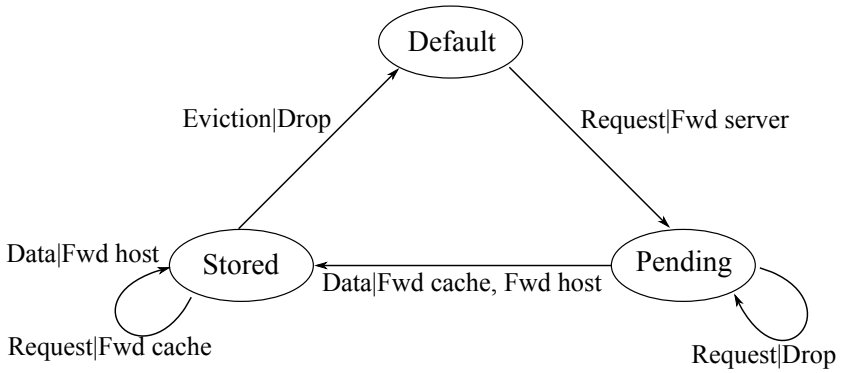


Fig. 3.4 State machine diagram for NDN packet forwarding. The notation on the arrow is event|action.

described in Fig. 3.4. Note that the identification of Data packets or Interest packets is performed assuming a single layer-2 network connecting server and hosts, but it can be easily generalized to layer-3 networks. To lookup and update the state, we use the content fingerprint as key for the state table. Since the content fingerprint is stored in the UDP ports of the packet, we set the lookup_scope and the update_scope equal to the UDP source and destination ports.

As an example, Table 3.2 shows the state table corresponding to a pending content, whose fingerprint corresponds to UDP ports 888 and 777. Notably, while Table 3.1 is fixed and pre-configured during bootstrapping, Table 3.2 stores the active flows (i.e., stored and pending contents) and its occupancy varies with the time.

3.4.2 S/N-DN Node using OpenState

We extend the basic NDN packet forwarding in OpenState for one host, described in the previous section, to implement the full functionality of the S/N-DN node with multiple hosts. As shown in Fig. 3.5, we have an OpenState switch connected with the cache, through the cache agent, at port P_C. The server is connected to the switch through port P_S, and the switch contains U user ports from P_1 to P_U. The stateful switch combines two state machines: XFSM 1 is responsible to forward the traffic to the cache or the server, mimicking the behavior of PIT and FIB of a standard NDN

Table 3.1 Flow table for NDN packet forwarding

Match fields		**Actions**	
State	**Event**	**Action**	**Next state**
Default	MACsrc=* MACdst=*	Fwd to server	Pending
Pending	MACsrc=server MACdst=*	Fwd to cache Fwd to host	Stored
Pending	MACsrc=* MACdst=*	Drop	Pending
Stored	MACsrc=cache MACdst=EVICTED	Drop	Default
Stored	MACsrc=cache MACdst=*	Fwd to host	Stored
Stored	MACsrc=* MACdst=*	Fwd to cache	Stored

Table 3.2 Example of state table in an S/N-DN node

Flow key	**State**
*	Default
UDP dst port = 888 UDP src port = 777	Pending
UDP dst port = ... UDP src port = ...	...

node; XFSM 2 is used for learning the MAC addresses of the hosts connected at the U user ports and guarantees layer-2 connectivity.

Each of XFSM 1 and XFSM 2 contains a state table and a flow table; thus, in our application we have four tables: state table 1, flow table 1, state table 2 and flow table 2. The flow table of XFSM 1 extends the flow table described earlier in Table 3.1. For simplicity, in the following we discuss in details the case with two user ports P_1 and P_2 (i.e., $U = 2$). Later, we will comment on how to generalize it to an arbitrary U. The flow entries of the two tables are shown in Tables 3.3 and 3.4, in decreasing order of priority. The "pending" states, referred to a specific content, are coded as:

- "Pending10": the request received from P_1 is pending;
- "Pending01": the request received from P_2 is pending;
- "Pending11": the requests received from P_1 and P_2 are pending.

Upon arrival of an Interest packet at XFSM 1, the content state is looked up from state table 1. Assuming that the content has been requested for the first time, the content is in the "default" state and the Interest packet is forwarded to the server (i.e., entries 1 and 2 in Table 3.3). If the arrival port is P_1 (or P_2), then the next state of the flow is "Pending10" (or "Pending01").

Moreover, the Interest packet is sent to XFSM 2 where the switch operates the standard MAC learning and forwarding procedure, following the approach proposed in [7]. The corresponding flow table is shown in Table 3.4. The lookup_scope of XFSM 2 is the packet destination MAC address, which is used to look up the state of the flow from state table 2. The next state depends on the ingress port of the incoming packet, i.e., if the in-port is P_1, then the next state is "portP1". The update_scope of XFSM 2 is the source MAC address of the packet. Thus, state table 2 stores the correspondence between the MAC address and the corresponding switch port.

Coming back to XFSM 1, entries 3–8 of Table 3.3 correspond to a content in "pending" state, i.e., the content has been forwarded to the server and the switch is waiting for the server response. Meanwhile, if another request arrives for the same content, the "pending" state will be updated to reflect the actual set of ports from which the requests arrived. Entries 9–11 of Table 3.3 manage the response from the server carrying the Data packet with the requested content. The switch forwards the content to the ports coded in the "pending" state and to the cache, and then changes the state of the content into "stored". As a result, any future request for this content will be forwarded to the cache instead of the server, coherently with entry 14. Then, the cache agent will answer with the requested content, using the destination MAC address corresponding to the requesting host. When such Data packet enters the switch, entry 13 will allow the correct forwarding to the destination port through XFSM 2. Finally, when the content is removed from the cache, the cache agent informs the switch to change the state of the corresponding flow from "stored" to "default" (entry 12).

3.4.3 S/N-DN node for $U > 2$

We discuss here how to generalize XFSM 1 for a generic number of user ports U. The "pending" state is now coded as "Pending$X$" where X is a binary string with 1 in position i whenever the Interest for a specific content is pending from port i. The

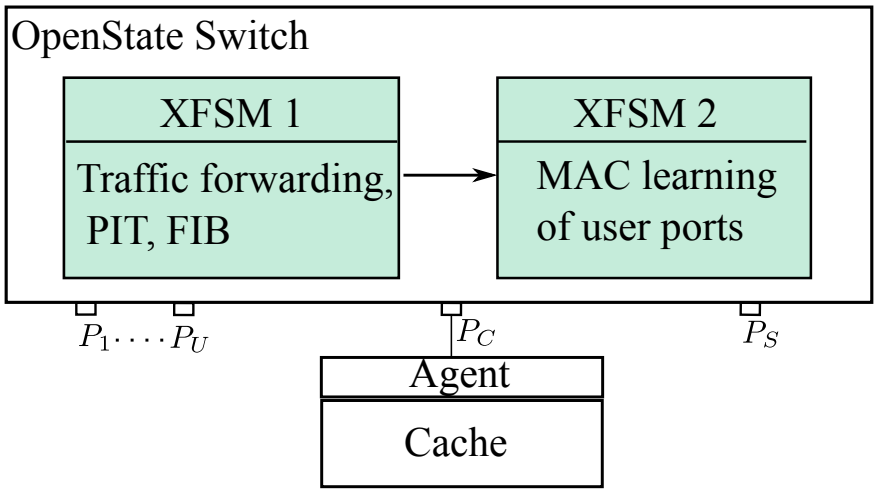


Fig. 3.5 Architecture of a stateful S/N-DN node.

total number of pending states is $2^U - 1$. For example, "Pending1010" means that the request is pending from user ports 1 and 3. Now, entries like 1–2 in Table 3.3 must be equal to U, i.e., one entry for each user port. Entries like 3–8 must be one for each pair of user port and "pending" state, thus summing to a total of $U(2^U - 1)$ entries. Entries like 9–11 in Table 3.3 must be one for each "pending" state, thus summing to $(2^U - 1)$. Finally, entries 12–14 in Table 3.3 will remain the same.

Table 3.5 summarizes the number of entries, which grows as $O(U2^U)$. The number of entries in state table 1 corresponds to the number of contents in "stored" or "pending" state. Thus, it is bounded by the maximum number of contents in the cache (equal to C) plus the number of contents that are "pending". Notably, one additional entry is present in the state table for the "default" state. Instead, the number of entries in state table 2 is equal to the number of MAC addresses learned by the switch, which is upper bounded by $U + 3$ assuming one MAC address learned for each user port, one entry for the cache, one entry for the server and one default entry.

Table 3.3 Flow table 1 of S/N-DN node, U=2

No.	Match fields		Actions	
	state	**event**	**action**	**next state**
1	Default	MACsrc=*, MACdst=*, in-port=P_1	Fwd to server, Goto XFSM2	Pending10
2	Default	MACsrc=*, MACdst=*, in-port=P_2	Fwd to server, Goto XFSM2	Pending01
3	Pending10	MACsrc=*, MACdst=*, in-port=P_1	Goto XFSM2	Pending10
4	Pending10	MACsrc=*, MACdst=*, in-port=P_2	Goto XFSM2	Pending11
5	Pending01	MACsrc=*, MACdst=*, in-port=P_1	Goto XFSM2	Pending11
6	Pending01	MACsrc=*, MACdst=*, in-port=P_2	Goto XFSM2	Pending01
7	Pending11	MACsrc=*, MACdst=*, in-port=P_1	Goto XFSM2	Pending11
8	Pending11	MACsrc=*, MACdst=*, in-port=P_2	Goto XFSM2	Pending11
9	Pending10	MACsrc=server, MACdst=*	Fwd to cache, Fwd on port P_1	Stored
10	Pending01	MACsrc=server, MACdst=*	Fwd to cache, Fwd on port P_2	Stored
11	Pending11	MACsrc=server, MACdst=*	Fwd to cache, Fwd on ports P_1,P_2	Stored
12	Stored	MACsrc=cache, MACdst=EVICTED	Drop	Default
13	Stored	MACsrc=cache, MACdst=*	Goto XFSM2	Stored
14	Stored	MACsrc=*, MACdst=*, in-port=*	Fwd to cache, Goto XFSM2	Stored

Table 3.4 Flow table 2 of S/N-DN node, U=2

Match fields		Actions	
state	**event**	**action**	**next state**
Default	in-port=P_1	Drop	portP1
Default	in-port=P_2	Drop	portP2
portP1	-	Fwd on port P_1	-
portP2	-	Fwd on port P_2	-

Table 3.5 Number of entries in XFSM 1 and XFSM 2

Table	State transition or mapping	Num. entries
Flow table 1	Default → Pending	U
	Pending → Pending	$U(2^U-1)$
	Pending → Stored	2^U-1
	Stored → Stored	2
	Stored → Default	1
Flow table 2	Default → PortP	U
	PortP → PortP	U
State table 1	Content → Stored	$C+1$
State table 2	MAC → PortP	$U+3$

3.4.4 Enhanced flow table definition in XFSM 1

Table 3.5 highlights the limited scalability of the flow table definition in XFSM 1 when U is large. This is mainly due to the self transition from/to "pending" state, whenever the state of a pending content must be updated to include the port from which a new request has been received. In other words, if the current state for a content is "pending$X$", with X being the adopted bit string representation of the arrival ports, and a new Interest packet for the same content arrives from port i, the new state "pending$Y$" is obtained by setting the ith bit of Y to one, i.e., using the logical OR operation:

$$Y = X \vee 2^{U-i+1} \qquad i = 1,\ldots,U. \tag{3.1}$$

Thus, to support this simple state transition in flow table 1, the controller has to pre-configure all the possible transitions from a given "pending$X$" state and a given i arrival port to the new state "pending$Y$", producing $U(2^U-1)$ entries as discussed before. This approach is what we described in Sec. 3.4.3 and it is actually what we have experimented since it is the only option available when adopting the OpenState switch emulator provided in [39]. Nevertheless, the original paper on OpenState [7] envisions the possibility of computing simple operations on the current state in order to compute the new state. In this case, (3.1) could be coded into a single entry in flow table 1, substituting all the $U(2^U-1)$ entries from "pending" → "pending". Similarly, with a single entry, it would be possible to configure all the "default" → "pending" transitions.

Table 3.6 Enhanced version of XFSM 1

Table	State transition	Num. entries
Flow table 1	Default → Pending	1
	Pending → Pending	1
	Pending → Stored	1
	Stored → Stored	2
	Stored → Default	1

Furthermore, if OpenState was able to operate a programmable action based on the current state, it would be possible to avoid also the $2^U - 1$ entries "pending" → "stored" states. Indeed, given "pending$X$" state, when a Data packet arrives from the server, the set of user ports where to forward the packet (in addition to the cache) could be defined as the set of destination ports where the ith bit of X is 1. This would allow to code the transitions "pending" → "stored" into just 1 single entry.

In summary, Table 3.6 shows the final number of entries if the programmability of OpenState would be allowed in terms of simple logical operations to compute both the new state and the actions to apply. The number of entries in the flow table would be $O(1)$ allowing the approach to scale for a large number of user ports.

3.5 Performance evaluation methodology

Here we introduce the methodology we use to evaluate the performance of our system. We start by presenting the performance metrics we adopt, then we describe the stateless approach for implementing the S/N-DN node, against which we benchmark the performance of our stateful approach. Finally, we describe our testbed and evaluation settings.

3.5.1 Performance metrics

In order to evaluate the system performance, we look at the following metrics:

- *End-to-end (e2e) delay*, measured in ms: time difference between the generation of an Interest packet and the arrival of the Data packet with the requested content at the host;

- *Cache download probability*, $P_{cache} \in [0,1]$: computed as fraction of content requests that are satisfied by directly downloading the content from the local cache;
- *Control traffic*, measured in bit/s: amount of traffic exchanged between the OF switch and the SDN controller over time, considering only the messages related to our NDN application;
- *Memory occupancy*, measured in kbytes: memory needed to store the state entries and flow entries in the switch.

3.5.2 Stateless caching in SDN

As a benchmark for our approach, we consider an S/N-DN node implemented through a standard, stateless OF switch. In the following, we use the terms *stateful approach* and *stateless approach* to represent, respectively, stateful and stateless data plane implementations of the S/N-DN node.

The architecture of a stateless S/N-DN node is shown in Fig. 3.6. An OF switch consists of U user ports, one port P_C connected to the cache through the cache agent, and one port P_S to connect to the server. The switch is controlled by the SDN controller, which stores the NDN-specific data structures, i.e., PIT, FIB and CLT. In addition, the controller is also responsible to forward the traffic towards the cache or the server and to run a reactive MAC learning application. To learn the MAC address of a host connected to a port, upon receiving the first packet from the host, the controller installs a flow entry in the OF switch, which is used to forward the subsequent packets to the host. Note that, in a network of stateless S/N-DN nodes, each of the OF switches is attached to a local cache but only one SDN controller stores the PIT, CLT and FIB tables for all the S/N-DN nodes.

The sequence of exchanged packets is shown in Fig. 3.7. The scenario and the packet formats are the same as in Fig. 3.3, i.e., the host requests content A twice, before and after the content is stored in the cache. In more detail, first, the host sends the Interest (P1) for content A to the switch, which then sends an OF packet_in message to the controller carrying a copy of P1. The controller creates an entry in the PIT, indicating that content A is needed by the host port. Then, it sends a packet_out message to the switch to forward the Interest (P1) to the server port. Hence, P1 is

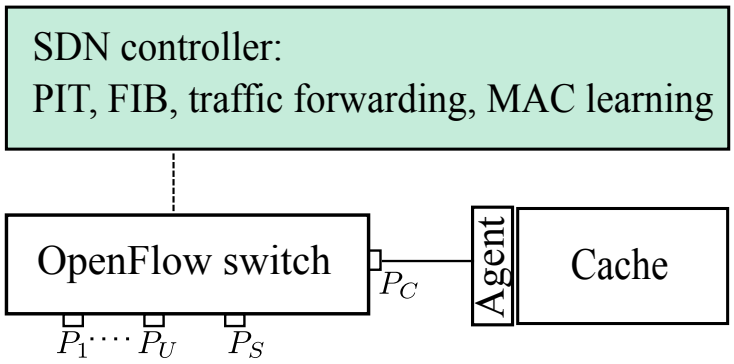
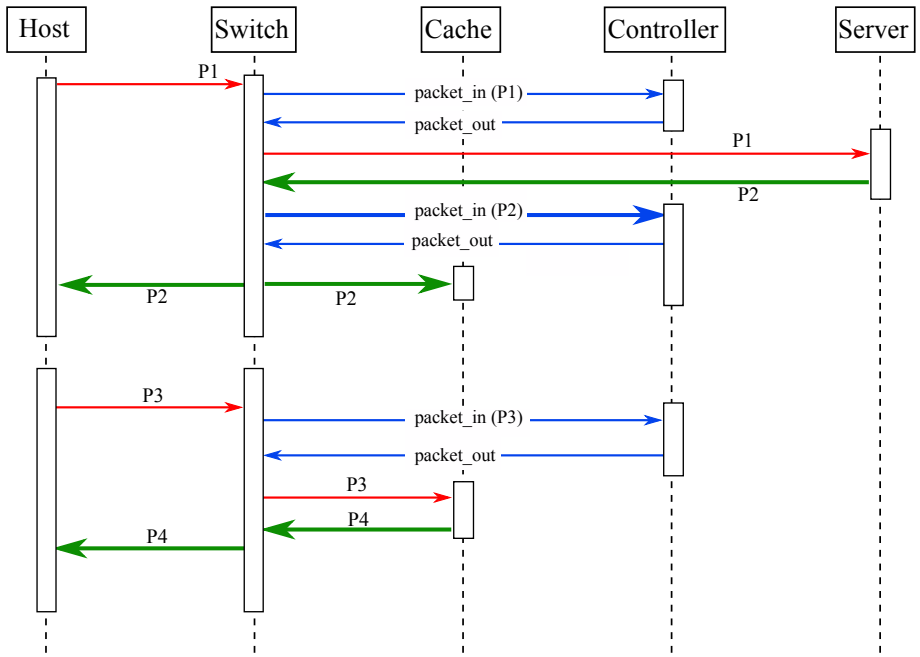


Fig. 3.6 Architecture of a stateless S/N-DN node.

forwarded to the server that sends back the Data packet (P2) carrying content A. Upon receiving the content, the switch again contacts the controller by sending out a packet_in message carrying P2. The controller now checks its PIT table and then instructs the switch, by sending a packet_out message, to send the content to the host port and to the cache port. After the previous step, content A has reached the host and it is stored in the cache. Now, if the host requests again the same content (P3), then a copy of the Interest packet is again forwarded to the controller in a packet_in message to check in the CLT if the content is available in the cache. Since now the content is in the cache, the controller instructs the switch, through a packet_out message, to forward the Interest (P3) to the cache port. The cache agent responds with the Data packet (P4) directed to the host MAC address. This packet is received by the switch and then forwarded to the host based on the local MAC matching rules. In addition, when a content is evicted from the cache, the cache agent sends an eviction message to the controller, such that the controller updates the corresponding CLT.

In the above scenario, the whole packet exchange, excluding the packets between the switch and the controller, are the same as those depicted in Fig. 3.3 for our proposed stateful approach. Thus, the two approaches are equivalent from a functional point of view. The stateless approach requires that the switch interacts with the controller for each Interest packet, since the main NDN data structures (PIT and CLT) are managed by the controller and not directly by the switch as in our stateful S/N-DN node.



	Ethernet	IP	UDP ports	
P1	Eth(Host)-->Eth(Server)	IP(Host)-->IP(Server)	hash(A)	Interest(A)
P2	Eth(Server)-->Eth(Multicast)	IP(Server)--> IP(Multicast)	hash(A)	Data(A)
P3	Eth(Host)-->Eth(Server)	IP(Host)-->IP(Server)	hash(A)	Interest(A)
P4	Eth(Cache)-->Eth(Host)	IP(Server)--> IP(Host)	hash(A)	Data(A)

Fig. 3.7 Packet interaction in stateless implementation of S/N-DN node when content A is requested.

3.5.3 Testbed implementation

We create our testbed on a server installed with Ubuntu 16.04. Mininet[2] is used to emulate our network scenario. We use OpenState virtual switch provided in [39] for stateful S/N-DN node and a standard OpenFlow v1.3 virtual switch for stateless S/N-DN node. The switch is managed by Ryu controller. The applications running on Ryu to support the stateful and stateless S/N-DN nodes have been developed in Python. Furthermore, the server and the cache agent (including the cache) are implemented as Mininet hosts running their respective software written in Python using Scapy[3] tool. The server can receive content requests and respond with actual

[2]http://www.mininet.org
[3]http://www.secdev.org/projects/scapy/

content data. The cache agent is able to: (i) save content received from the server in a local Least Recently Used (LRU) cache, (ii) respond to content requests received from the host by sending the content data, and (iii) inform the switch about eviction of any content from the cache. We use a reference NDN traffic generator[4] to generate a trace of NDN Interest packets. This trace of content requests is replayed by the host at a specified rate using the `tcp_replay` Linux utility.

3.5.4 Evaluation of performance metrics

In order to evaluate the memory occupancy of the flow tables, we use the standard OpenFlow "flow_stats" request and reply messages. The SDN controller sends the flow_stats request message and receives back the memory occupied by each of the installed flow entries. In the stateful approach, since the flow tables do not change after the initial configuration, the flow_stats request and reply messages are exchanged only once in the beginning.

For the evaluation of the control traffic, we use Wireshark[5] to count the number of control messages (exchanged between the OF switch and the SDN controller) as well as to compute the size of each message. Furthermore, we only consider the control messages required for content retrieval; we do not include the messages required for initial configuration or the ones for retrieving flow statistics.

3.5.5 Evaluation scenario

We setup the scenario described in Fig. 3.2, comprising of an OpenState/OpenFlow switch, a cache, a server and a host. The switch is configured by the application running on top of the Ryu controller. The server and the cache run their respective software, while the host replays the traffic generated by the NDN Traffic Generator. A total of 1000 content requests are periodically generated at the rate of 1 packet/s. This guarantees that the time interval between two subsequent requests is always greater than the e2e delay of the first request, regardless of the stateful/stateless approach. The content catalog contains 100 data items, each of fixed size. The content for each Interest packet is chosen at random across the catalog according to

[4]https://github.com/named-data/ndn-traffic-generator
[5]https://www.wireshark.org/

Table 3.7 Scenario parameters

Parameter	Description
d_{ss}	One-way delay between switch and server [ms]
d_{sc}	One-way delay between switch and controller [ms]
$\hat{C}$	Normalized cache size

a given distribution of content popularity. The system parameters are summarized in Table 3.7. Therein, d_{ss} represents the one-way propagation delay between the OF switch and the server, while d_{sc} represents the one between the OF switch and the SDN controller. Furthermore, $\hat{C} \in [0,1]$ represents the normalized cache size, i.e., the size C of the cache normalized by the catalog size, both measured in terms of content items.

3.6 Experimental Results

Given the aforementioned evaluation settings, we perform experiments on our testbed to validate the proposed approach and assess its performance. We now assume uniform distribution of content popularity and data size[6] equal to 1200 bytes. Fig. 3.8 shows the end-to-end (e2e) delay of the content requested by the host, for $\hat{C} = 1$ and $d_{ss} = 100$ ms. In the case of our stateful approach, d_{sc} does not affect the e2e delay experienced by the host, since, once the switch has been configured, it never interacts with the controller. The figure clearly shows that there are two phases in the system: a transient phase during which the cache is not full and the switch stores a copy of each of the distinct content items in the cache, and a stationary phase which starts as soon as the cache is fully utilized. During the transient phase, a requested content not stored in the cache is fetched from the server and the incurred e2e delay is around 218 ms, which is the round-trip time between the switch and the server plus the processing time. Conversely, during the stationary phase, all of the 100 contents are available in the cache and the e2e delay is always around 18 ms, which is coherent with the processing time evaluated in the transient phase.

In the following we will present our results during the stationary phase.

[6]Data size comprises also the standard NDN header for content identification.

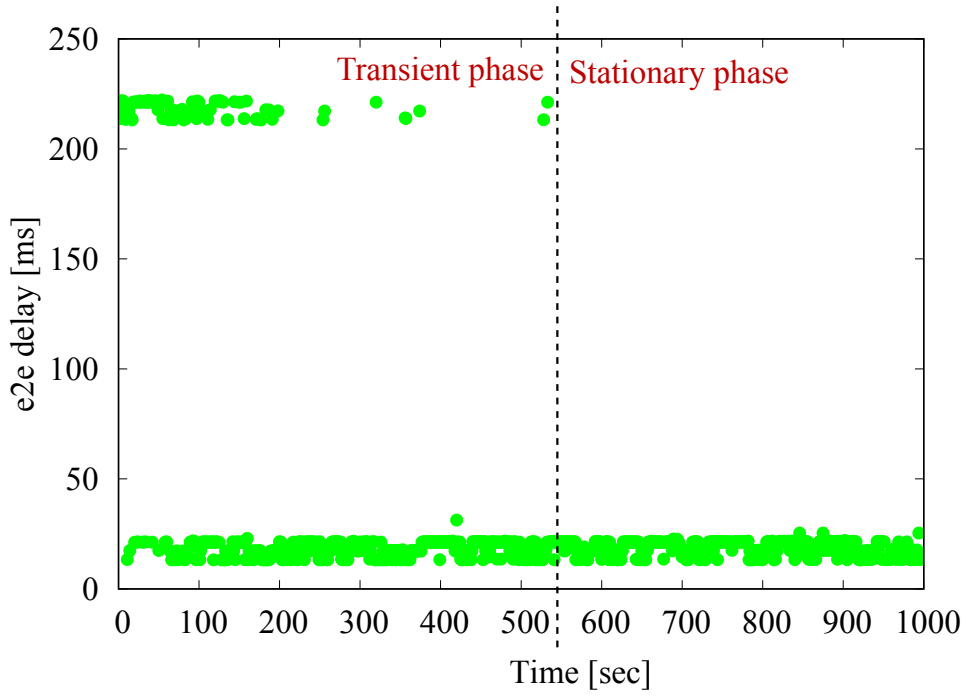


Fig. 3.8 The e2e delays during transient and stationary phases for the stateful approach, when $\hat{C} = 1$ and d_{ss}=100 ms.

3.6.1 End-to-end delays

Fig. 3.9 depicts the e2e delays of the content requested by the host for different values of d_{ss}. The results refer to the stateful approach with $\hat{C} = 0.5$. At $d_{ss} = 100$ ms, for each of the requested content, the e2e delay is centered around either 18 ms or 218 ms, depending on the availability of the content in the cache. Indeed, even if the cache is always fully utilized in the stationary phase, it cannot store all of the 100 contents due to its limited size. Thus, some contents are evicted to make room for newly requested items, according to the LRU eviction policy. Similarly, at $d_{ss} = 50$ ms, the e2e delays are centered around either 18 ms or 118 ms. Finally, at $d_{ss} = 0$ ms, even the content that is not in the cache is retrieved instantly from the server, thus the e2e delays is centered around 18 ms.

Fig. 3.10 instead compares the stateful and the stateless approaches in terms of the distribution of the e2e delay. In particular, it shows stacked bar plots, for different normalized cache sizes, representing the probability that the e2e delay lies within the specified intervals. In the case of stateful approach, the e2e delay is distributed in the intervals 0–99 ms and 200–299 ms, depending on the availability of

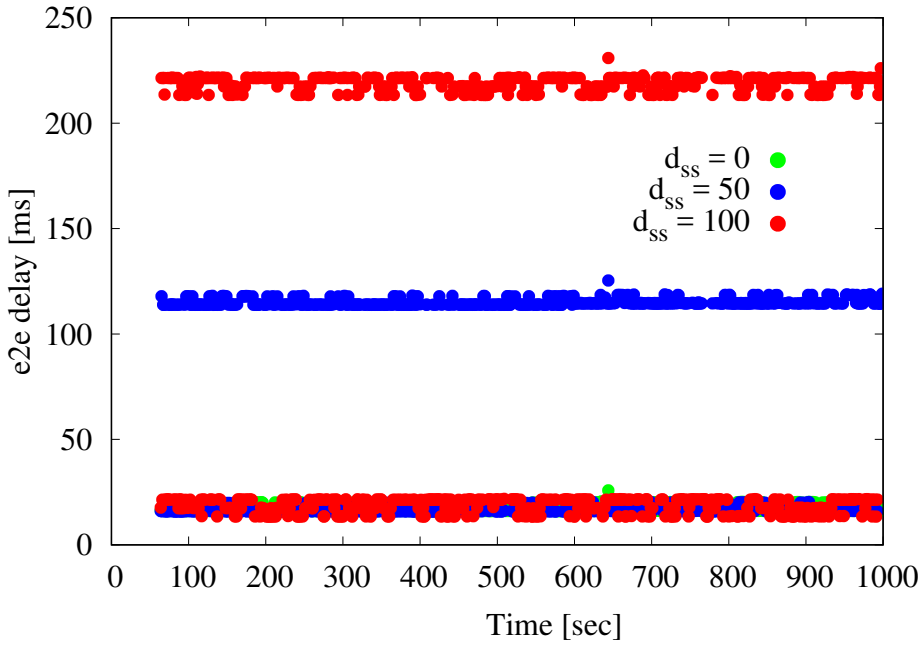


Fig. 3.9 e2e delays during the stationary phase, for the stateful approach and $\hat{C} = 0.5$.

the content in the cache. For the stateless approach, instead, much larger e2e delays are observed: they now range between 200 and 299 ms, and between 600 and 699 ms. This is because the stateless approach relies on the communication with the Ryu controller to implement the functionality of the S/N-DN node, which depends on the parameter d_{sc}. Moreover, the impact of the normalized cache size is also evident for both approaches. At $\hat{C} = 0.9$, approximately 93% of the content requests are satisfied by the cache, hence incurring lower e2e delays, while for the remaining 7% requests the contents are fetched from the server resulting in much larger e2e delays. At $\hat{C} = 0.5$, approximately 50% of the content requests are satisfied by the cache, while at $\hat{C} = 0.1$, approximately 9% and 91% of the content requests are satisfied, respectively, from the cache and the server.

Fig. 3.11 compares the average e2e delay, computed over all content requests, for the stateful and stateless approaches, as a function of $\hat{C}$. The delay of the stateful approach is approximately three times less than that of the stateless approach for small $\hat{C}$, while the difference becomes two times for large $\hat{C}$. We plot the average e2e delay as a function of d_{sc} in Fig. 3.12. The figure clearly shows that the average delay increases linearly with d_{sc} for the stateless approach, while the performance is independent of parameter d_{sc} in the stateful case. Indeed, recall that, unlike the latter,

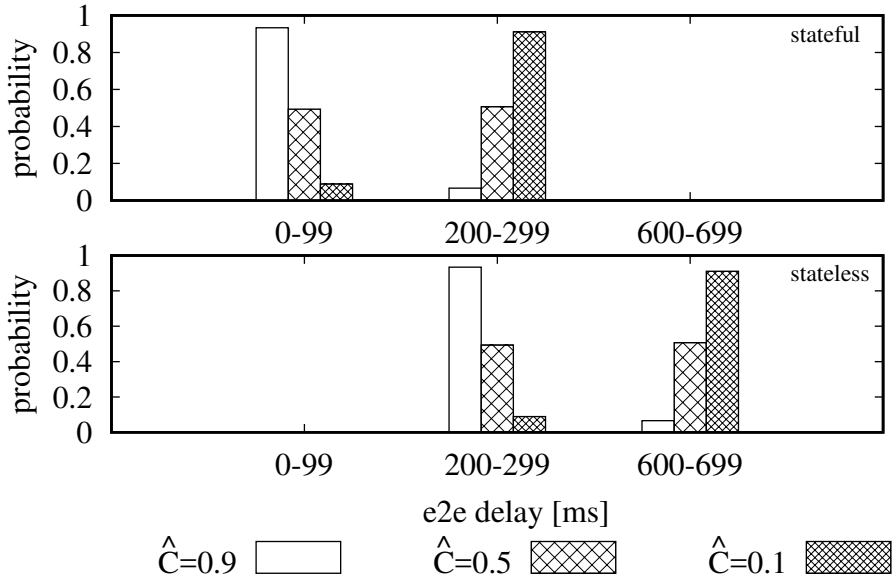


Fig. 3.10 Probability density function of the e2e delay for stateful and stateless approaches, for d_{ss}=100 ms and d_{sc}=100 ms.

the stateless solution heavily relies on the communication with the SDN controller in order to implement the NDN functionalities, hence its performance significantly depends on the communication latency between the controller and the switch.

3.6.2 Cache download probability

We compute the cache download probability as $\hat{C}$ varies as well as for content requests that are generated according to uniform and Zipf (with $\alpha = 0.75$) distributions of content popularity. Moreover, the behavior of the cache download probability does not depend on the adopted (stateful/stateless) approach, rather on the generation process of the content requests. This is because, as previously discussed in our setup, the time to retrieve a requested content is much lower than the time interval between two requests. As shown in Fig. 3.13, the cache download probability exhibits a linear behavior for the uniform popularity, and non-linear for Zipf popularity; both behaviors are well known and in accordance with Che's approximation for LRU caches [54].

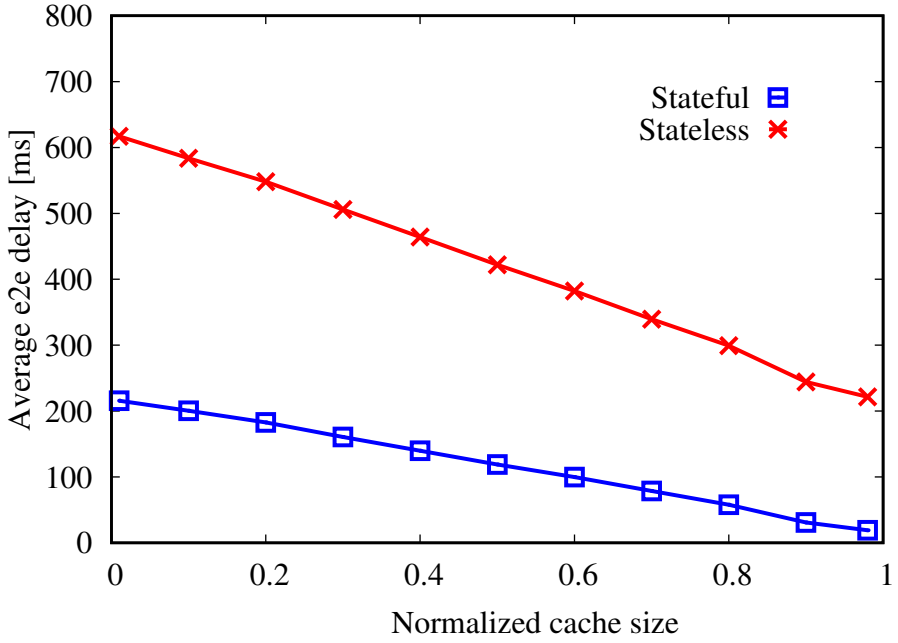


Fig. 3.11 Average e2e delay for d_{ss}=100 ms and d_{sc}=100 ms.

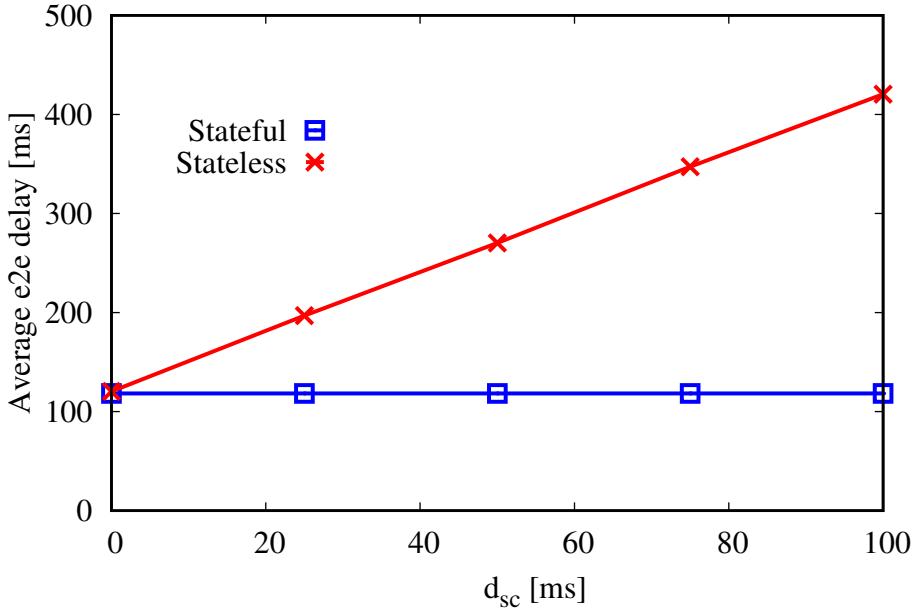


Fig. 3.12 Average e2e delay for d_{ss}=100 and $\hat{C} = 0.5$.

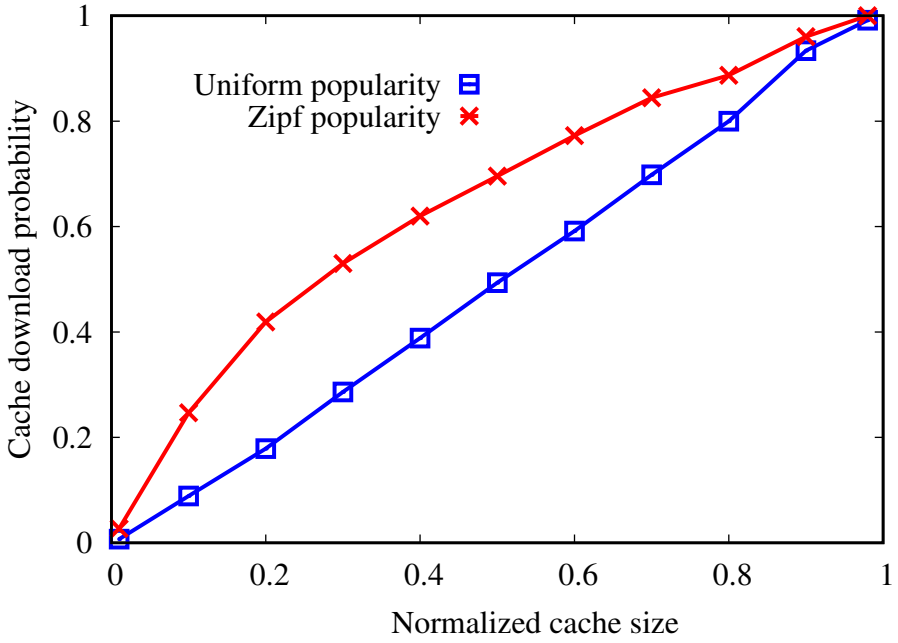


Fig. 3.13 Cache download probability for d_{ss}=100 ms and d_{sc}=100 ms.

3.6.3 Control traffic for stateless approach

The OF control traffic is essential in the stateless data plane implementation of the S/N-DN node, as explained in Sec. 3.5.2. In particular, 4 OF messages (see Fig. 3.7) are exchanged between the switch and controller when the requested content is not available in the cache. In addition, while operating in the stationary phase, the cache is full and each unavailable content, after being fetched from the server, is stored in the cache in place of the least recently used content. This leads to an additional control message sent to the controller informing about the evicted content, thus a total of 5 OF messages are exchanged. On the other hand, only 2 OF messages are exchanged when the content is available in the cache. Furthermore, note that the installation of the flow entry for the destination MAC address of the host takes place during the transient phase, hence it is not included in this computation. In summary, in our scenario, the average number of control messages for the stateless approach can be computed as follows:

$$5(1-P_{cache})+2P_{cache}. \tag{3.2}$$

Fig. 3.14 shows the average number of control messages per content request, for different values of $\hat{C}$, under uniform and Zipf distributions of content popularity. At small $\hat{C}$, around 5 OF messages are exchanged, since the corresponding P_{cache} is close to 0. As $\hat{C}$ increases, P_{cache} increases and thus the number of messages decreases. If the cache stores all content items (i.e., $\hat{C} = 1$), then $P_{cache} = 1$ and only 2 OF messages are exchanged.

In addition to evaluate the number of control messages, we also measure the corresponding bandwidth. Table 3.8 shows the empirical size of the Ethernet frames of the OF messages that are exchanged when the requested content is available in the cache and when it is not. We consider two possible cases: either the carried data is small (equal to 100 bytes) or large (equal to 1200 bytes, almost the maximum allowed to avoid IP fragmentation in the OF traffic).

Table 3.9 shows the network overhead for the two approaches. The network overhead is defined as the OF control traffic exchanged with the controller in stationary conditions, i.e., when the cache is full, normalized by the data traffic received from the server. In the stateless approach, the network overhead depends on the availability of the content in the cache as well as the size of the data packets, since a copy of the data packet may be sent to the controller carried by an OF packet_in message. When the data is small and is not locally cached, the overhead can reach almost 800%, since more than 1.1 kbytes (refer to the size of the corresponding 5 OF messages in Table 3.8) are exchanged for just 100 bytes of data. The overhead is instead much smaller when the data is large, since the OF traffic is better amortized. Nevertheless, in the best case, i.e., the large data locally available at the cache, the overhead is still approximately 30%.

We also show, in Fig. 3.14, the average bandwidth required for the stateless approach for each content request, and the required bandwidth for the large data packets, assuming 1 request per second. The required bandwidth varies between 3 kbps and 18 kbps, which is small due to the limited request rate considered in the experiment. Notably, this depends on the request generation rate and the cache size.

In our proposed stateful approach, on the other hand, after the initial preconfiguration, the switch acts autonomously without any interaction with the SDN controller (excluding the legacy monitoring messages), thus the number of OF messages as well as the network overhead is null.

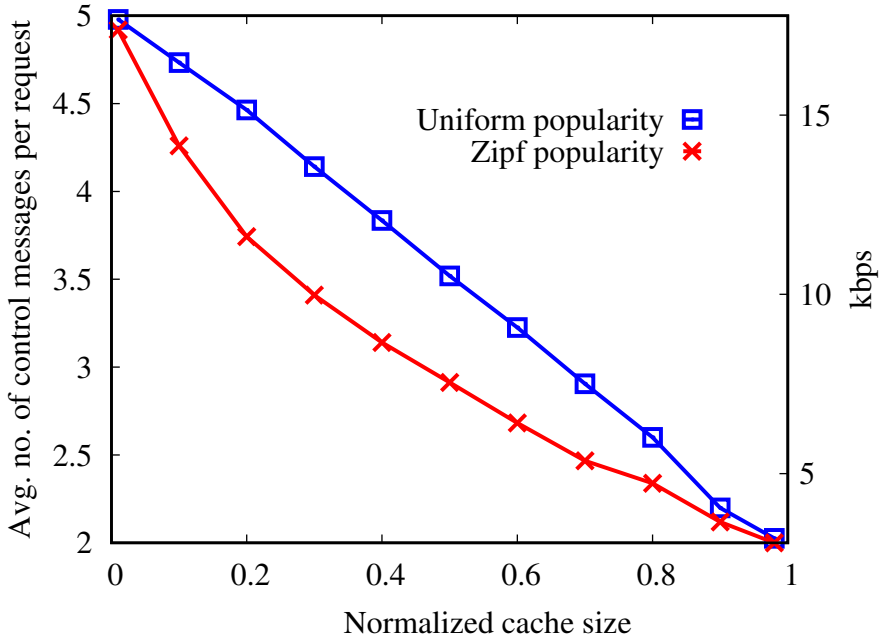


Fig. 3.14 Control traffic for the stateless approach, for d_{ss}=100 ms, d_{sc}=100 ms and one request per second.

Table 3.8 Ethernet frame size of OF messages for data between 100 and 1200 bytes

Scenario	**OF message**	**Size**
Content not available	packet_in(Interest)	276 bytes
	packet_out(Interest)	106 bytes
	packet_in(Data)	354-1454 bytes
	packet_out(Data)	122 bytes
	packet_in(eviction)	276 bytes
Content available	packet_in(Interest)	276 bytes
	packet_out	106 bytes

Table 3.9 Control traffic overhead for the stateful and stateless approaches

Approach	**Scenario**	**Data packet**	**Normalized OF traffic**
Stateful	any	any	0%
Stateless	Content not available	100 bytes	799%
		1200 bytes	180%
	Content available	100 bytes	269%
		1200 bytes	31%

Table 3.10 Empirical memory occupancy for XFSM 1 and XFSM 2

Table	State transition	Memory (per-entry)
Flow table 1	Default → Pending	168 bytes
	Pending → Pending	152 bytes
	Pending → Stored	176 or 192 bytes
	Stored → Stored	152 or 160 bytes
	Stored → Default (eviction)	152 bytes
Flow table 2	Default → PortP (MAC learning)	128 bytes
	PortP → PortP (MAC forwarding)	88 bytes

3.6.4 Memory occupancy

Table 3.10 shows the empirical memory required for each entry in the flow tables of XFSM 1 and XFSM 2. Instead, state tables are not present in standard OF architectures, hence no message is available to query their memory occupancy. In the following results we estimate their occupancy by using as reference: (i) $32+U+1$ bits for each entry of state table 1, obtained as 32 bits (i.e., the content fingerprint) plus U bits to code the set of destination ports plus 1 additional bit for the stored state, and (ii) $48+U$ bits for each entry of state table 2 (i.e., 48 bits of MAC address that serves as flow key in XFSM 2 plus U bits to code the state).

Fig. 3.15 shows the total memory occupancy for both the flow tables and the state tables, respectively, assuming $C = 100$ contents. Since the number of flow entries grows as $O(U2^U)$ as discussed in Sec. 3.4.3, the memory occupancy of the flow tables increases rapidly by increasing the number of user ports U. However, it can be observed that the memory occupied by the flow tables inside the switch for $U = 8$ is approximately 350 kbytes, which is feasible for hardware implementation based on TCAM memories [55]. Regarding the state entries, they depend on the user ports as well as the cache size (see Table 3.5) but their occupancy remain below 0.6 kbyte, thus they are negligible.

With regard to the stateless approach, the whole functionality of the NDN node is implemented in the SDN controller while the switch plays a minor role; as a result, the number of flow entries installed in the switch is limited. Specifically, only U entries are installed, one for each MAC address associated with each user port. Note, however, that the small memory occupancy required by the stateless approach comes at the cost of large e2e delays.

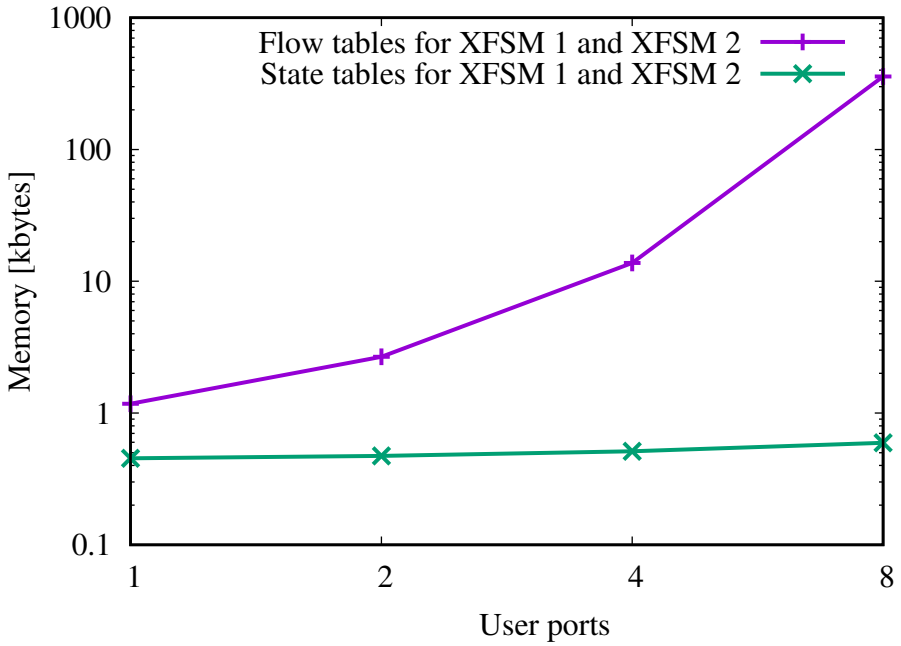


Fig. 3.15 Empirical memory occupancy in the OpenState switch for a stateful S/N-DN node.

3.7 Results for $U > 1$

When number of user ports $U > 1$, and each host connected to one user port sends a sequence of content requests uncorrelated with that of the other hosts, then, from the switch/cache point of view, the combined requests from all the hosts are seen as a single stationary process with a given distribution (e.g., uniform). Consequently, the system achieves similar performance in terms of average delay, cache download probability and average control traffic. Moreover, the performance (excluding the memory occupancy) is primarily a function of the cache size, not the sequence of content requests. For $U = 2$, the average delay and cache download probability are shown in Figs. 3.16 and 3.17 respectively.

3.8 Related Works

NDN is an ICN architecture whose baseline implementation was carried out under the project called Content Centric Networking (CCN) [37]. The significance of

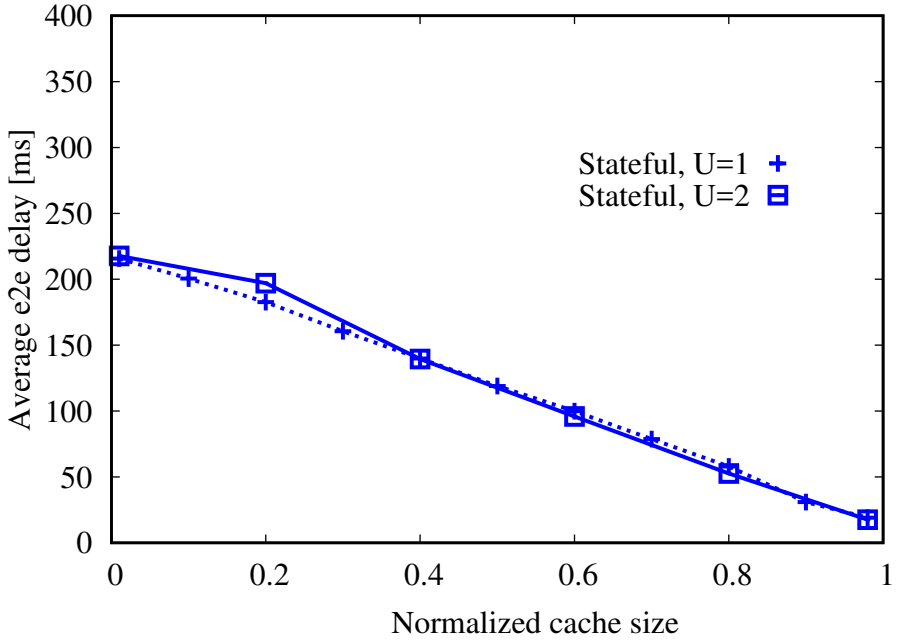


Fig. 3.16 Average e2e delay for d_{ss}=100 ms, d_{sc}=100 ms and U=2.

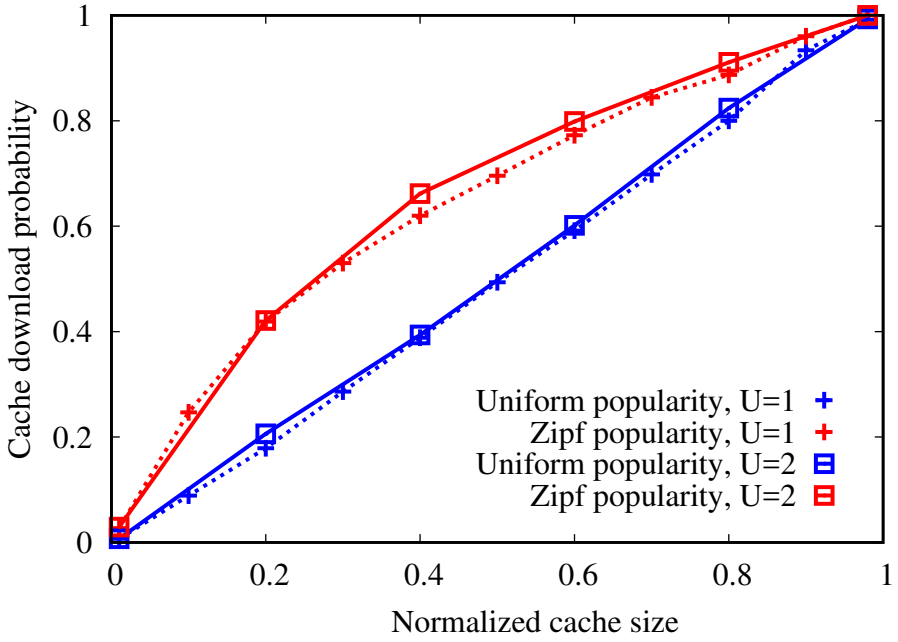


Fig. 3.17 Cache download probability for d_{ss}=100 ms, d_{sc}=100 ms and U=2.

deploying SDN and ICN capabilities in future networks has led to a considerable amount of research that aims at combining both the technologies.

A first family of works implements ICN over SDN, without an integration among the two layers. The works in [56, 50] implement an OF-based CCN node, in order to provide CCN functionalities such as caching and name forwarding in an SDN-based data plane. The CCN node consists of an OF switch, a wrapper and a CCNx daemon (i.e., a reference CCN implementation). The wrapper pairs each port of the switch with a CCNx interface, thus making it responsible for the communication between the switch and the daemon. A similar approach is taken in NDNFlow [57] where an ICN node is implemented by installing CCNx daemon on a legacy OF switch. The authors consider a network scenario involving both ICN-enabled as well as legacy OF switches. They implement an ICN module in the SDN controller, which uses an ad-hoc protocol on the south-bound interface of the controller, in parallel to OpenFlow, to control the ICN-enabled OF switch. In order to implement a CCNx node, computing resources must be integrated with the switch to run the CCNx daemon. Instead, in our stateful S/N-DN node the processing of NDN packets runs directly on the switch (thanks to the supported state machines) and thus allows to achieve full line-rate performance.

In [51] and [58], switches are not ICN-aware, making the SDN controller responsible for managing ICN request and response packets. This solution is similar to the stateless approach we compare with, and incurs large delays due to the required interaction between the switch and the controller, differently from our stateful approach which completely avoids such interaction. Very recently, other works have investigated how to integrate ICN in a 5G architecture using a stateless SDN approach. Indeed, [59] provides a 5G-ICN architecture implemented with the network slicing paradigm, while [60] combines NDN and SDN in the context of vehicular networks. Due to the centralized nature of approach, both works suffer limited reactivity as perceived by the users due to the continuous interaction between the controller and the switch to forward properly data and interest packets. The work in [61] proposes to combine ICN, C-RAN and SDN in heterogeneous networks. The adopted approach is based on logically centralized controllers responsible for all the ICN related operations (e.g., content addressing and matching), thus incurring large processing overload on the controllers and limiting large-scale deployment of heterogeneous networks. The paper suggests to install matching rules directly on the SDN switches to reduce the interaction with the controllers. This enhanced version is similar to the stateless approach considered in our work.

The work [62] is one of the initial works that combine SDN with ICN. Differently from our work, the considered ICN architecture has three main components: rendezvous, topology management and forwarding. The network function of topology management and rendezvous are realized in two centralized nodes, called topology manager and rendezvous server respectively. The topology manager builds forwarding identifiers, used for source routing, using bloom filter-based encoding scheme [63]. The forwarding network function is delegated to the SDN controller. This centralized approach for packet forwarding incurs significant scalability issues. In contrast, we focus on a decentralized approach according to which NDN Interest/Data packets are routed without involving the controller.

Similar to our idea of implementing an NDN node using a stateful SDN data plane, NDN.p4 [64] provides a preliminary implementation of NDN node using P4 abstraction [41] to program the data plane. In particular, [64] enables a P4 switch to process NDN Interest and Data packets. The switch implements PIT and FIB tables but cache lookup is not implemented. In our work, we adopt OpenState instead of P4, and we allow the switch to implement the PIT and CLT, thus it operates autonomously the forwarding of NDN traffic among cache, server and hosts. In addition, we evaluate the performance of the whole S/N-DN node considering relevant metrics such as latency, memory occupancy in the switch and control traffic.

The work most similar to ours is [65], which studies the feasibility of "breadcrumb" forwarding in OpenState switches. Such forwarding is a reverse path forwarding scheme, which maintains states for the opposite direction of a flow. The paper concentrates on a hardware proof-of-concept implementation based on FPGA. Moreover, the authors discuss few applications (e.g., CCN node, MPLS switching) which can be implemented by "breadcrumb" forwarding. In particular, the work describes the implementation of the PIT. The XFSM discussed in [65] for the PIT is very similar to our XFSM 1, proposed in Sec. 3.4.3, and thus suffers the same scalability limitations since the flow entries in the flow table grows as $O(U2^U)$. However, it does not support explicitly the content storage, and thus it does not provide the additional functionalities (as CLT, content eviction, automatic forwarding to cache or to the server, integration with the MAC learning/forwarding) that we support to fully enable the S/N-DN node.

Finally, in our work we do not investigate the effect of the specific caching policy adopted in NDN networks. We just consider a basic user-driven approach in which

the content requests from the users dictate the contents stored in the cache and the eviction is managed with a traditional LRU policy. More advanced policies can be devised to optimize the efficiency of the caching scheme. In this context, [66] discusses the challenges for content caching and routing to provide video streaming service in ICN mobile wireless networks. The proposed video streaming solution takes into account content popularity to devise an efficient content caching strategy. Moreover, it provides a mechanism to improve content delivery by considering mobility of the nodes and selecting optimal content providers. In our work, we do not investigate the possible implementation of such schemes through a stateful approach, even if we expect that some of them could be easily implemented.

3.9 Summary

We proposed a novel solution to implement NDN leveraging the programmability of stateful SDN switches. Our architecture comprises stateful SDN switches, each of which is attached to a local cache. This combination of stateful switch and cache can successfully replicate the behavior of an NDN node, and we therefore referred to it as stateful S/N-DN node. We implemented the stateful S/N-DN node using OpenState, and a system testbed using Mininet, OpenState and Ryu SDN controller so as to evaluate the performance of our solution. We also benchmarked our stateful approach with a stateless data plane implementation of the S/N-DN node. We highlighted that, in a traditional stateless approach, the OpenFlow switch must rely on the communication with the controller in order to implement the functionality of the NDN node, thus resulting in large overhead and large end-to-end delays. On the contrary, our stateful approach does not need to involve the controller after the initial configuration of the OpenState switch, hence it yields zero control traffic and short latency.

Chapter 4

Conclusions

In this thesis, we study caching techniques to determine efficient ways to store contents in cache and control the cache nodes. These aspects are important to be investigated as next generation cellular networks, e.g., 5G systems, will include caching capabilities in the network.

Chapter 2 focusses on efficient provision of streaming data to connected cars as they drive along a road covered by cache-enabled wireless edge nodes. Our RICH prefetching policy uses the mobility information of the cars, i.e., dwell time of cars under coverage of edge nodes, to make prefetching decisions. Our scheme increases the probability to download contents from the cache, thus reducing the traffic to be fetched from the backhaul. In addition to devising a new policy, we discuss the system aspects of our solution and provide a complete system architecture as well as the governing protocol for the entities inside the architecture. One important entity in our system is a central Prefetcher module that determines the content of edge node caches using the dwell time distribution of cars. In order to test the performance of our solution, we use a realistic traffic dataset of the Italian city of Bologna and carry out simulations in OMNeT++. RICH is able to achieve significant improvements in terms of cache throughput, cache hit probability, cache occupancy, backhaul traffic and backhaul overhead with respect to both alternative mobility-agnostic and state-of-art mobility-aware strategies. Furthermore, we also evaluate the robustness of RICH by introducing errors in dwell time of car as well as in car path in terms of traversed ENs.

Chapter 3 discusses an efficient way to implement and control cache nodes in NDN leveraging the programmability of stateful SDN switches. By attaching a local cache with a stateful SDN switch, it is possible to replicate the behavior of an NDN node, hence we term it as stateful S/N-DN node. Our stateful approach is implemented using OpenState and the performance is evaluated using a system testbed comprising Mininet, OpenState and Ryu SDN controller. An alternative stateless approach of implementing S/N-DN node is used for the benchmarking purpose, which incurs large control traffic and large end-to-end delays. On the other hand, our stateful approach incurs short end-to-end delays and zero control traffic as the stateful SDN switch does not rely on the SDN controller for its functionality after the initial configuration.

References

[1] 5G empowering vertical industries. 5G-ppp White paper, Feb. 2016.

[2] 5G PPP use cases and performance evaluation models. 5G-ppp White paper, April 2016.

[3] 5G and media & entertainment. 5G-ppp White paper, January 2016.

[4] Vision on software networks and 5G. 5G-ppp White paper, Software networks WG, January 2017.

[5] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, kc claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. Named data networking. *SIGCOMM Comput. Commun. Rev.*, 44(3):66–73, July 2014.

[6] 5G: Challenges, research priorities, and recommendations. Joint white paper, NetWorld 2000, Jan. 2015.

[7] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. OpenState: Programming platform-independent stateful OpenFlow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, April 2014.

[8] Francesco Malandrino, Carla Chiasserini, and Scott Kirkpatrick. The price of fog: A data-driven study on caching architectures in vehicular networks. In *ACM IoV-VoI*, pages 37–42, July 2016.

[9] M. Fiore, C. Casetti, and C.F. Chiasserini. Caching strategies based on information density estimation in wireless ad hoc networks. *IEEE Transactions on Vehicular Technology*, 60(5):2194–2208, 2014.

[10] N. Le Scouarnec, C. Neumann, and G. Straub. Cache policies for cloud-based systems: To keep or not to keep. *IEEE CLOUD*, 2014.

[11] A. Gravey Li Zhe, G. Simon. Caching policies for in-network caching. *IEEE ICCCN*, 2012.

[12] Cesar Bernardini, Thomas Silverston, and Olivier Festor. Mpc: Popularity-based caching strategy for content centric networks. In *IEEE ICC*, pages 3619–3623, June 2013.

[13] S. Tewari and L. Kleinrock. Proportional replication in peer-to-peer networks. *IEEE INFOCOM*, 2006.

[14] Feixiong Zhang, Chenren Xu, Yanyong Zhang, KK Ramakrishnan, Shreyasee Mukherjee, Roy Yates, and Thu Nguyen. EdgeBuffer: Caching and prefetching content at the edge in the MobilityFirst future internet architecture. In *IEEE WoWMoM*, pages 1–9, June 2015.

[15] Sourav Kumar Dandapat, Swadhin Pradhan, Niloy Ganguly, and Romit Roy Choudhury. Sprinkler: distributed content storage for just-in-time streaming. In *ACM CellNet*, pages 19–24, June 2013.

[16] B. Blaszczyszyn and A. Giovanidis. Optimal geographic caching in cellular networks. In *IEEE ICC*, pages 3358–3363, June 2015.

[17] Mehdi Dehghan, Anand Seetharamz, Ting He, Theodoros Salonidis, Jim Kurose, and Don Towsley. Optimal caching and routing in hybrid networks. In *IEEE MILCOM*, pages 1072–1078, Oct 2014.

[18] Laura Bieker, Daniel Krajzewicz, AntonioPio Morra, Carlo Michelacci, and Fabio Cartolano. Traffic simulation for all: a real world traffic scenario from the city of Bologna. In *Modeling Mobility with Open Data*, pages 47–60. Springer, 2015.

[19] ETSI EN 302 663 v1.2.1 Intelligent Transport Systems (ITS); Access layer specification for intelligent transport systems operating in the 5 GHz frequency band. *ETSI Standard*, 2013.

[20] ETSI TS 103 319 broadband radio access networks (bran); 5 GHz high performance RLAN; mitigation techniques to enable sharing between RLANs and road tolling and intelligent transport systems in the 5 725 MHz to 5 925 MHz band. *Manuscript in preparation*.

[21] Khan Mohammad Irfan and Jérôme Härri. Can IEEE 802.11p and WI-FI coexist in the 5.9GHz ITS band? In *IEEE WoWMoM*, June 2017.

[22] Cooperative Intelligent Transportation Systems (C-ITS) Platform. Jan 2016.

[23] OMNet++ discrete event simulator.

[24] INET framework.

[25] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *IEEE INFOCOM*, volume 1, pages 126–134, Mar 1999.

[26] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of caching strategies in modern cellular backhaul networks. In *ACM MobiSys*, pages 319–332, June 2013.

[27] Junchao Ma, Jiahuan Wang, and Pingzhi Fan. A cooperation-based caching scheme for heterogeneous networks. *IEEE Access*, 2016.

[28] Zheng Chen, Jemin Lee, Tony QS Quek, and Marios Kountouris. Cooperative caching and transmission design in cluster-centric small cell networks. *IEEE Transactions on Wireless Communications*, 2017.

[29] Jon Froehlich and John Krumm. Route prediction from trip observations. In *SAE Technical Paper*. SAE International, April 2008.

[30] John Krumm. A Markov model for driver turn prediction. In *SAE Technical Paper*. SAE International, April 2008.

[31] Pralhad Deshpande, Anand Kashyap, Chul Sung, and Samir R Das. Predictive methods for improved vehicular WiFi access. In *ACM MobiSys*, pages 263–276, June 2009.

[32] Dipankar Raychaudhuri, Kiran Nagaraja, and Arun Venkataramani. MobilityFirst: a robust and trustworthy mobility-centric architecture for the future Internet. *ACM SIGMOBILE Mobile Computing and Communications Review*, 16(3):2–13, 2012.

[33] Zhou Su, Yilong Hui, and Qing Yang. The next generation vehicular networks: a content-centric framework. *IEEE Wireless Communications*, 24(1):60–66, 2017.

[34] Junaid Ahmed Khan and Yacine Ghamri-Doudane. Saving: socially aware vehicular information-centric networking. *IEEE Communications Magazine*, 54(8):100–107, 2016.

[35] Dennis Grewe, Marco Wagner, and Hannes Frey. Perceive: Proactive caching in ICN-based vanets. In *IEEE VNC*, pages 1–8, Dec 2016.

[36] Giulia Mauri, Mario Gerla, Federico Bruno, Matteo Cesana, and Giacomo Verticale. Optimal content prefetching in ndn vehicle-to-infrastructure scenario. *IEEE Transactions on Vehicular Technology*, 66(3):2513–2525, 2017.

[37] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. In *ACM CoNEXT*, pages 1–12, December 2009.

[38] 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.

[39] OpenState SDN project home page.

[40] S. Pontarelli, M. Bonola, G. Bianchi, A. Capone, and C. Cascone. Stateful OpenFlow: Hardware proof of concept. In *IEEE HPSR*, pages 1–8, July 2015.

[41] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[42] Protocol independent switch architecture.

[43] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, August 2013.

[44] Yongmao Ren, Jun Li, Shanshan Shi, Lingling Li, Guodong Wang, and Beichuan Zhang. Congestion control in named data networking – a survey. *Computer Communications*, 86(Supplement C):1 – 11, 2016.

[45] Cheng Yi, Alexander Afanasyev, Ilya Moiseenko, Lan Wang, Beichuan Zhang, and Lixia Zhang. A case for stateful forwarding plane. *Computer Communications*, 36(7):779 – 791, 2013.

[46] G. Carofiglio, M. Gallo, and L. Muscariello. ICP: Design and evaluation of an interest control protocol for content-centric networking. In *Proceedings IEEE INFOCOM Workshops*, pages 304–309, March 2012.

[47] N. Rozhnova and S. Fdida. An effective hop-by-hop interest shaping mechanism for CCN communications. In *Proceedings IEEE INFOCOM Workshops*, pages 322–327, March 2012.

[48] Yaogong Wang, Natalya Rozhnova, Ashok Narayanan, David Oran, and Injong Rhee. An improved hop-by-hop interest shaper for congestion control in named data networking. *SIGCOMM Computer Communication Review*, 43(4):55–60, August 2013.

[49] Giovanna Carofiglio, Massimo Gallo, and Luca Muscariello. Joint hop-by-hop and receiver-driven interest control protocol for content-centric networks. *SIGCOMM Computer Communication Review*, 42(4):491–496, September 2012.

[50] Xuan Nam Nguyen, Damien Saucez, and Thierry Turletti. Providing CCN functionalities over OpenFlow switches. Research report, August 2013.

[51] S. Salsano, N. Blefari-Melazzi, A. Detti, G. Morabito, and L. Veltri. Information centric networking over SDN and OpenFlow: Architectural aspects and experiments on the OFELIA testbed. *Computer Networks*, 57(16):3207 – 3221, 2013. Information Centric Networking.

[52] A. Ooka, S. Ata, T. Koide, H. Shimonishi, and M. Murata. OpenFlow-based content-centric networking architecture and router implementation. In *Future Network Mobile Summit*, pages 1–10, July 2013.

[53] S. Eum, M. Jibiki, M. Murata, H. Asaeda, and N. Nishinaga. A design of an ICN architecture within the framework of SDN. In *IEEE ICUFN*, pages 141–146, July 2015.

[54] Hao Che, Ye Tung, and Zhijun Wang. Hierarchical web caching systems: modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305–1314, Sep 2002.

[55] SDN system performance.

[56] Xuan Nam Nguyen, D. Saucez, and T. Turletti. Efficient caching in content-centric networks using OpenFlow. In *IEEE INFOCOM*, pages 67–68, April 2013.

[57] N. L. M. van Adrichem and F. A. Kuipers. NDNFlow: Software-defined named data networking. In *IEEE NetSoft*, pages 1–5, April 2015.

[58] M. Vahlenkamp, F. Schneider, D. Kutscher, and J. Seedorf. Enabling ICN in IP networks using SDN. In *IEEE ICNP*, pages 1–2, Oct 2013.

[59] R. Ravindran, A. Chakraborti, S. O. Amin, A. Azgin, and G. Wang. 5G-ICN: Delivering ICN services over 5G using network slicing. *IEEE Communications Magazine*, 55(5):101–107, May 2017.

[60] S. H. Ahmed, S. H. Bouk, D. Kim, D. B. Rawat, and H. Song. Named data networking for software defined vehicular networks. *IEEE Communications Magazine*, 55(8):60–66, 2017.

[61] C. Yang, Z. Chen, B. Xia, and J. Wang. When ICN meets C-RAN for HetNets: an SDN approach. *IEEE Communications Magazine*, 53(11):118–125, November 2015.

[62] D. Syrivelis, G. Parisis, D. Trossen, P. Flegkas, V. Sourlas, T. Korakis, and L. Tassiulas. Pursuing a software defined information-centric network. In *2012 European Workshop on Software Defined Networking*, pages 103–108, Oct 2012.

[63] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. LIPSIN: Line speed publish/subscribe inter-networking. *SIGCOMM Comput. Commun. Rev.*, 39(4):195–206, August 2009.

[64] S. Signorello, R. State, J. François, and O. Festor. NDN.p4: Programming information-centric data-planes. In *IEEE NetSoft*, pages 384–389, June 2016.

[65] G. Bianchi, M. Bonola, and S. Pontarelli. On the feasibility of "breadcrumb" trails within OpenFlow switches. In *IEEE EuCNC*, pages 122–127, June 2016.

[66] C. Xu, P. Zhang, S. Jia, M. Wang, and G. M. Muntean. Video streaming in content-centric mobile networks: Challenges and solutions. *IEEE Wireless Communications*, 24(5):157–165, October 2017.

Appendix A

Publications and Awards

Journals

- A. Mahmood, C. Casetti, C. F. Chiasserini, P. Giaccone and J. Harri, "Efficient caching through stateful SDN in named data networking", *published in Transactions on Emerging Telecommunications Technologies*, November 2017. **This paper is presented in Chapter 3**.
- A. Mahmood, C. Casetti, C. F. Chiasserini, P. Giaccone and J. Harri, "The RICH prefetching in edge caches for in-order delivery to connected cars", *submitted in IEEE Transactions on Vehicular Technology*, April 2018. **This paper is presented in Chapter 2**.

Book Chapter

- C. Fiandrino, P, Giaccone, A. Mahmood and L. Maioli, "Enriching remote control applications with fog computing", *published in CISIS*, July 2017.

Conference Proceedings

- A. Mahmood, C. Casetti, C. F. Chiasserini, P. Giaccone and J. Harri, "Mobility-aware edge caching for connected cars", *published in IEEE WONS*, January 2016.
- A. Bianco, P. Giaccone, A. Mahmood, M. Ullio and V. Vercellone, "Evaluating the SDN control traffic in large ISP networks", *published in IEEE ICC NGN*, June 2015.

Awards

- **Travel and participation grant award:** BMW summer school, 6-11 July 2015.