Content Distribution over Software Defined Networks

Ahmed Amine Loukili

POLYTECH NICE-SOPHIA

Université Nice Sophia Antipolis

POLYTECH NICE SOPHIA ANTIPOLIS

MASTER IFI/ UBINET TRACK

Content distribution over Software Defined Network

Internship Report presented for the degree of
Master of Science

DIANA Team
INRIA Sophia Antipolis.
France, August 2016.

Author :

Ahmed Amine LOUKILI

Supervisors :

- Damien SAUCEZ
- Thierry TURLETTI

Abstract

As internet usage has grown recently, people today consume and produce large amounts of media on the internet and in ways very different from before. The notion of a 'destination site' is on the decline since in this digital age we enter a search query and trust that a complicated algorithm will provide us with the most valuable source of content on that topic. In parallel, the advent of virtualization and network function softwarization pushed Software Defined Networking to become an emerging technology making networks programmable thus enabling network operators to automate their network management. Using the SDN approach to cope with the recent network user needs and to view the network as a program needs the deployment of APIs (Application Programming Interface). In this context, we proposed, implemented, and, tested a Network as a Service (NaaS) API. This API abstracts the network and devices operations to simplify their management, and to provide functionalities offered by the SDN architecture.

In this work we focused on a particular application: content distribution over a SDN network. More specifically, we have implemented an API to represent the entire network and to place a copy of the most popular content in the caching unit close to the client in order to balance the load and increase the performance of the network. The goal of this work is to prove that content distribution and network management can both be done without needing a radical change in the network architecture, therefore we implemented a proof-of-concept leveraging, OpenFlow controllers, HTTP proxy, and a RESTful API.

Table of content

List of Figures

List of Tables

Chapter 1: Introduction

1. Problem description

1.1. Context

Network usage has evolved to be dominated by content dissemination [1], the increasing demand for highly scalable and efficient distribution of content has motivated the networking community to start investigating new Internet architectures to cope with this evolution. One approach of these architectures is called Information Centric networking (ICN) [2], the key aspects of ICN is to treat the content as a primitive, decoupling location from identity, security, and access, and retrieving content by name. The ICN approach advocates a redesign of the internet network layer through a major shift from host-to-host communication model to a content-based one, ICN propositions and implementations, for instance CCN [3] (Content Centric Network) moves the universal component of the network from IP to chunks of named content. Such architectures can be qualified as a long-term solution for the content distribution problem due to the complexity and challenges that their deployment implies.

Software defined networking (SDN) [4], which is an emerging networking paradigm that aims at making the network programmable, offers the possibility to implement optimizations that previously were theoretical in nature due to the implementation complexity. SDN main aspects include the separation of the control plane from the data plane, a well-defined interface between the data and control plane, and a logically centralized software based controller that gives a network view for the control and management of network applications. SDN introduced new functionalities both for networking and computational point of

view that are usually provided through a Network as a Service (NaaS) Application Programming Interface (API) [5].

In this context, the ANR DISCO (Distributed SDN Controllers for rich and elastic network services) project [6] has been launched with the objective to bring flexibility, scalability, and resiliency in SDN architectures, but also, richness and elasticity for the deployment of network services. DISCO investigates in particular the urbanization of virtual network functions or network appliances (e.g., load-balancer, deep packet inspection, ciphers), that can be embedded into virtual machines, thus consuming communized CPUs, memory, and network capabilities. DISCO also studies and develops a NaaS API enabling network programmability.

1.2. Objectives

The goal of this work is to design, implement, and deliver a proof of concept of the DISCO's NaaS API to provide a complete abstraction of the network. The API allows us to get a global view of the network status and also allows to change its behavior. For example, the NaaS API can be used to place the contents in the right place of the network based on some criteria (e.g., using the popularity of the content) in order to maximize the network performance and enhance the network usability for end users. In other words, this API will be a tool to deploy and manage a content distribution service over SDN, it will provide a solution to link the application layer with the network layer to ease the management of such service.

The deployment of a content distribution service involves making routing policies, packet inspection because we have communication between two different layers, monitoring to know who consumes what and also handling mobility in case of content caching, this is the reason why we need to use SDN architecture that allows the programmability of the network where we can program each node the way we want using APIs.

The main objectives of this work can be summarized as follows:

- **Design and implementation of a REST API**: this API that will be used for the deployment of the content distribution service.
- **Testing and validation for the content distribution scenario**: to make sure that the API fulfils the desired operations related to the content distribution service (content placement, topology discovery, content caching…)
- **testbed design and benchmarking**: to have all the metrics related to the evaluation setup that allows to predict the behaviour of the network.
- **Benchmarking the network performances of the proposed solution**: this part is important to see the impact of the proposed solution on network performances.

2. Related Work

Numerous projects have been established for studying and proposing new architectures/solutions for the content distribution problem. A very interesting work [7] was done investigating and presenting a proof-of-concept design of an incrementally deployable ICN architecture. The main idea behind this work is to adopt an evolutionary approach instead of the clean-slate approach [8] that most of the ICN architectures propose, which means that this work presents some minimal and attainable changes that allow to achieve the same benefits of ICN (e.g., performance, mobility and security).

Incrementally deployable ICN architecture or idICN is the name of this new architecture achieved by using the widely used techniques and technologies from the past decade, requiring small changes to hosts or their protocols. This solution is based on an end-to-end mechanism (i.e., implemented at the edge of the network) to get the key performances of ICN. However, idICN involves some changes to the current internet that can be qualified as a limitation and potential stumbling blocks of the solution. Such kind of solutions always know a blocking

point where they face the reality and complexity of implementing changes on existing networks in a large scale, this is why the networking community started converging toward solutions based on virtualization and network function softwarization since the deployment of such kind of technologies has known a noticeable improvement and maturity.

Many proposals were presented adopting the SDN approach, a very interesting one comes with the idea of inserting a content-management layer in the SDN controller [9], this solution is called ContentFlow, which is a network architecture leveraging the principles of SDN to achieve the ICN goal of placing content at the center of the network: namely, with ContentFlow, a centralized controller in a domain will manage the content, resolve content to location, enable content-based routing and forwarding policies, manage content caching, and provide the extensibility of a software controller to create new content-based network mechanisms. The overall vision of ContentFlow is to have a transparent caching network that can be placed between a service provider network and a consumer network, it proposes also a content management layer installed in the controller that uses HTTP header information to identify content, routes it based on name and maps it back to TCP and IP semantics so that the whole system can operate on an underlying legacy network without any modification to either clients or servers. OpenWeb [10] was also proposed as a solution leveraging Openflow protocol and HTTP proxies with caching capabilities, with the idea of implementing cache systems operating at the layer-2 (cache systems are generally accessed through layer 4-7 scripts and commands) which makes it easy for administrators to introduce the system just by inserting it into the network and clients can use it transparently.

As we can notice, the SDN approach offers more flexibility and possibilities to come up with new proposals for content management using the current internet architecture and technologies, offering the same benefits as the ICN approach offers with minimal changes and less complexity which makes those

systems easy to maintain, evolve and scale. In the same direction, our work finds inspiration in these proposals but presents a different approach based on API exploitation and content popularity.

3. Outline

In chapter 2 an overview of the different technologies related to API implementation and SDN architectures is provided. A brief definition is presented for each technology then we explain the choices that we have made. Chapter 3 provides a definition of the NaaS API that we implemented to manage the content distribution service. In chapter 4 we can see in detail the implementation elements and steps then we introduce the evaluation setup and present the evaluation results in chapter 5. We finish with the conclusion in chapter 6.

Chapter 2: Technological Choices

This chapter has for goal to present and compare the different technologies related to the API implementation and explain each technological choice we have made.

1. Communication paradigm

Since we are developing a network API, it means that we will have to define the way service provider and consumer will interact, we have investigated these two communication paradigms and made our choice:

1.1. Representational State Transfer (REST):

REST (Representational state transfer) [11] is an architectural style, and an approach to communication that is often used in the development of web services. The REST style emphasizes that interactions between clients and services is enhanced by having a limited number of operations (verbs). Flexibility is provided by assigning resources their own unique Uniform Resource Identifiers (URIs). In order to be REST (we say a REST API or RESTful API) there are six constraints that the API must adhere to. The goal of these constraints is to maximize the scalability, independence and interoperability of software interactions. These six constraints are:

1 – Client / Server: This constraint must exist to maximize the portability of server-side functions to other platforms. With SDN, this usually means that completely different applications, even in different languages, can use the same functions in a REST API. The “applications” would be the client, and the controller would be the “server”.

2 – Stateless: Rest adds a constraint to the Client-Server interaction, the communication must be stateless in nature, such that each request from client to

server must contain all the information necessary to understand the request. All state is kept client-side, the server does not retain any record of client state. This constraint leads to more scalability, reliability and visibility.

3 – Caching: Cache constraint is about storing some data that can be used later in the client side in order to eliminate some interactions, improve efficiency, scalability and user-perceived performance by reducing the average latency of a series of interactions. This constraint implies also the possibility to have inconsistency between the Cache and the Server.

4 – Uniform Interface: This Constraint distinguishes the REST architecture from other network-based styles, information is transferred in a standardized form independently of the application that generated it. With this approach the system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.

5 – Layered System: REST introduces the layered system constraint, which allows a system to be comprised of multiple architectural layers, this is done by constraining the components behavior such that each component cannot "see" beyond the layer it can interact with. The layers can be used for different purposes we mention the following: encapsulate legacy services; protect new services from legacy clients; improve system scalability by enabling load balancing.

6- Code-On-Demand: This is an optional REST constraint, allowing client functionalities to be extended by downloading and executing code in the form of applets or scripts.

1.2. Simple Access Protocol (SOAP)

SOAP[12] is an XML-based messaging protocol that allows programs running on different operating systems to communicate, it is a standard for

encoding messages in XML that invoke functions in other applications. The SOAP approach requires writing or using a server program (to serve data) and a client program (to request data). In SOAP there is a rigid contract between client and server, everything is expected to break if either side changes anything, constant updates following any change (e.g., if any change happens on the client side an update should be triggered to inform the server of this change) is needed, and the client needs previous knowledge on everything he will be using, or he could not communicate with the server.

1.3. Technology adopted

In the context of our project, we have chosen to use the REST communication paradigm because it offers high scalability and better performance when providing services to a large number of clients in high speed networks. The REST implementation is lightweight, does not leverage much bandwidth, which makes it a better fit to our purpose.

Once we have chosen the communication model to adopt we had to decide which technology to use to transport our communications, in the following we present our choice:

✓ **REST over HTTP:**

We can apply REST to other protocols, as long as we ensure the stateless property but we decided to use HTTP because its implementation is lightweight, available on a wide range of architectures, open, and human-readable. To implement REST over HTTP there are some rules to follow:

- **Use URI as resource identifier:** REST is based on URI in order to identify resources, so an application should build its URI accurately, taking into account the REST constraints.
- **HTTP verbs such as identifying transactions:** The second rule of a REST architecture is to use existing HTTP verbs rather than including the operation in the URI of the resource (POST , GET …)

- **HTTP responses as resource representation:** it is important to have in mind that the answer sent is not a resource but it is the representation of a resource.

Note that there is also a big difference between a RESTful API and a HTTP API. A RESTful API adheres all the REST constraints. An HTTP API is any API that makes use of HTTP as their transfer protocol. This means that even SOAP can be considered as an HTTP API, as long as it will use HTTP for transport.

2. Data representation

After defining the way to interact with our API and how to transport the communications we had to choose the appropriate data representation format to use, we present the following choices:

2.1. JavaScript Object Notation (JSON)

JSON [13] is a lightweight data representation format that is simple, flexible and easy to understand. It is built on two structures:

- A collection of name/value pairs.
- An ordered list of values.

JSON can represent four primitive types (strings, numbers, Booleans, and null) and two structures types (objects and arrays).

2.2. Extensible Markup Language (XML)

XML [14] is a general-purpose specification for creating custom markup languages, it specifies the standards with which you can define your own markup languages with their own set of tags. XML dialects are the default file format for many office-productivity software, including Microsoft Office, OpenOffice, AbiWord, and Apple's iWork.

2.3. Technology adopted

We decided to use the JSON data representation, because it is compact, lightweight, and easy to be processed by computers and humans. JSON introduces less overhead than XML and thus it offers better performances.

3. API Implementation frameworks

Now that we have made the different technological choices mentioned before, we need to start implementing our REST API, for that we need a framework that will offer more flexibility and simplicity in the implementation. Here are two examples of frameworks:

3.1. FLASK

Flask [15] is a web development microframework written in Python[1], while it is a microframework, it does not contain a lot of features, but still it offers very interesting tools like: template engine (Jinja2), RESTful request dispatching, flexibility. Flask aims to keep the core simple but extensible, it does not oblige the user to use a specific technology, such as what database to use. Flask also gives the freedom to structure your application according to your needs, it has only few predefined requirements, the routing system is simple yet powerful in Flask all you have to do is to use decorators for each view to define the routes.

3.2. Django

Django [16] is a full-stack web framework written in python and that has definite structure and conventions. The primary goal of Django is to ease the creation of complex, database-driven applications. Django emphasizes reusability of components, rapid development, and the principle of “don’t repeat yourself “to avoid redundancy.

[1] We did not look for other programming languages because python was imposed by the project.

In order to use Django, we should be able to understand all the structure of the framework and adapt our application to this structure, unlike Flask, using Django would introduce more constraint in your application and make it more complex to achieve.

3.3. Technology adopted

Our choice was to use Flask instead of Django because of its simplicity, it allows to implement complex applications with a minimal number of lines of code and it has no dependencies. Moreover, it has few predefined conventions unlike Django that offers less freedom to the user in the development of the application.

4. SDN Controller

Since we are in a SDN context, a centralized approach is adopted based on the use of a SDN controller that has always a full view of the network and implements all the intelligence of the system while the other network components only implement elementary functions without any form of intelligence. In the following we present two well known SDN controllers:

4.1. OpenDayLight Open SDN controller

OpenDayLight [17] is an open source SDN controller hosted by the Linux foundation and is part of the OpenDayLight Project (ODL) that aims at enhancing software-defined networking (SDN) by offering a community-led and industry-supported framework for the OpenDaylight Controller. The controller is implemented within it's own Java Virtual Machine (JVM) which allow the controller to integrate any Hardware and operating system as long as it supports JAVA. It exposes Northbound APIs that are used by applications to fetch network information and push instructions to the controller to make changes, and it includes support for different SDN standards(e.g. Openflow[18]).

4.2. Floodlight Open SDN controller

Floodlight [19] Open SDN Controller is an enterprise-class, Apache-licensed, Java-based OpenFlow Controller. It uses the Openflow protocol to manage and orchestrate traffic flows in the network. Openflow is the first and the most used SDN standard, it defines the communication protocol that allows the controller to establish new configurations by speaking directly to the control plane.The Floodlight Controller can be advantageous for developers, because it offers them the ability to easily adapt software and develop applications and is written in Java. Included are representational state transfer APIs that make it easier to program interface with the product.

4.3. Technology adopted

Floodlight and OpenDaylight are both written in Java and very known, well-supported SDN controllers. They both include exposure with a REST API and a web based GUI. The only difference between the two is that OpenDaylight is not made only for Openflow networks since it supports non-OpenFlow southbound protocols, and since we are mainly interested in Openflow networks we have decided to work with the Floodlight Open SDN controller.

5. Data Storage:

We need to store information within our network in a distributed fashion, to that aim, we present the two following database examples that could be used in our implementation:

5.1. PostgreSQL database

PostgreSQL [20] is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness.

PostgreSQL prides itself in standards compliance. Its SQL implementation strongly conforms to the ANSI-SQL:2008 standard. It has full support for subqueries (including subselects in the FROM clause), read-committed and serializable transaction isolation levels. And while PostgreSQL has a fully relational system catalog which itself supports multiple schemas per database, its catalog is also accessible through the Information Schema as defined in the SQL standard.

5.2. MongoDB database

MongoDB [21] is a free and open-source cross-platform document-oriented database. Classified as a NoSQL [22] database, MongoDB avoids the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas (MongoDB calls the format BSON), making the integration of data in certain types of applications easier and faster. MongoDB is developed by MongoDB Inc. and is free and open-source, published under a combination of the GNU Affero General Public License and the Apache License. As of July 2015, MongoDB was the fourth most widely mentioned database engine on the web, and the most popular for document stores.

5.3. Technology adopted

We decided to use PostgreSQL 9.4 Database. This choice was essentially based on the power and flexibility of SQL it offers, and also for the effective features it has (e.g., support for Master Slave replication with high performance and scalability). As already mentioned we will be using Flask web Framework to implement our API, to ease the integration and manipulation of the database within our application we are using the very powerful toolkits SQLalchemy [25] and psycopg2 [26].

Chapter 3: The API

1. API Definition

As said before, the API is a crucial tool to deploy the content distribution service (since it allows to link the application and network layers), it is also simplifying operations on devices and modeling the network as an abstract directed graph where each node represents an entity or an operation of the network and where edges indicates the relationship between them. This abstraction hides the notion of network layers to focus on network operations and on contents.

In this section we will define each network node family, resources, see how we can give to each resource in the network a unique address to communicate with it and then describe the basic operations used by the API.

1.1. Definition of entities

The network is modeled by the API as an abstract graph, we can distinguish four families of nodes:

- **Compute:** a node of type compute is a node which functionality is related to processing operations, we have two types of compute nodes, **Processes** (e.g., thread) and **Processors** (e.g., CPU)
- **Network:** a node of type network is a node which functionality is related to networking operations, we have three types of network nodes, **Flows**, **Ports** and **Links**
- **Storage:** a node of type storage is a node which functionality is related to storage Operations, we have two types of storage nodes, **Content** (e.g., file) and **Store** (e.g., Cache).

In practice the different elements of a content distribution network are more than simple nodes of each node family, they are a composition of different entities that is why we define the slice node which define this composition:

- **Slice:** a node of type slice is a Meta node that is used to define a composition of different types of nodes (e.g., Router).

For example a cache can be at the same time a processing unit (Compute node) to which are attached some networking functionalities (Network node) and that stores content (Storage node), in this case we use the slice node to define this composition of nodes. By definition a node is always a slice on its own.

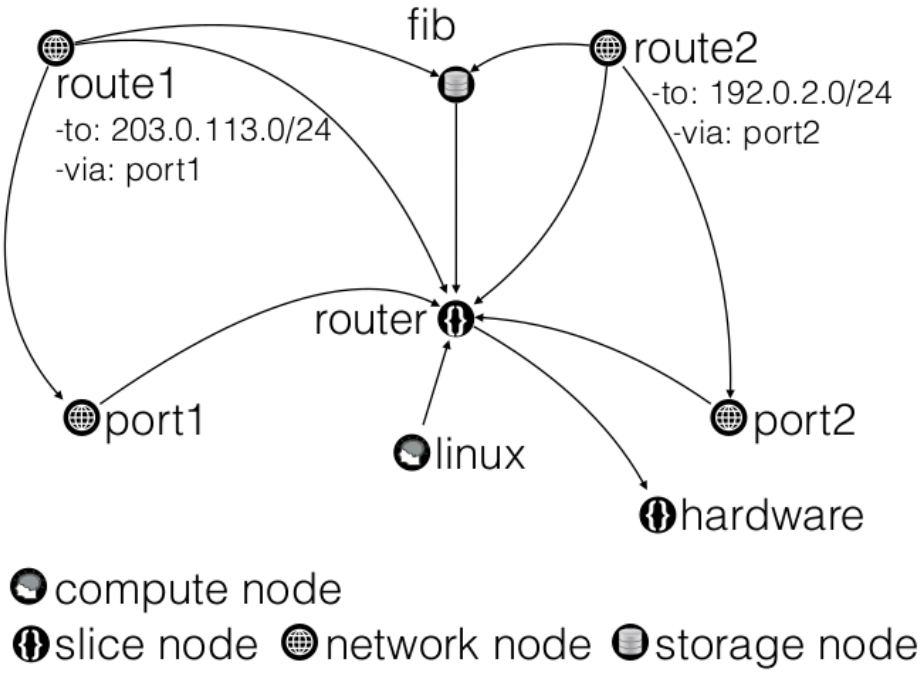

Figure 1 : Model of a two ports router with two routes.

Figure 1 shows a simple two-network-ports Linux router and its abstract graph representation. The router is composed of two network ports modeled as network nodes, the routing software modeled as a compute node, and the FIB modeled as a storage node. Each route is represented by a network node. All these independent nodes are linked to a slice representing the whole router that is itself linked to a slice that abstracts the actual hardware where Linux is installed. This way, migrating the router to another piece of hardware would just correspond to changing the link to be attached to another hardware in the model.

1.2. Definition of resources

A network resource in general refers to a form of data, information and hardware that can be accessed using a shared or private tool, within our network we are also using this notion of resources, which can be any kind of network node/component that we will interact with using the API, it can be a Cache, Server, Switch, Content… and each one of these resources is belonging to one or more family of nodes.

Identifying the different resources in the network helps in defining the addresses that the API will use to orchestrate the network. These addresses are called Uniform Resource Identifier (URI). We will see in the following their importance in our implementation and how they are constructed.

1.3. Construction of Uniform Resource Identifier (URI):

One of the most important properties of the REST communication paradigm that we are adopting is the definition of URIs that uniquely identify each resource of the network, the URI is used to reach the resource and interact with it using the different operations that we will see in the next section.

There is no general rule in the URI construction, it is just a naming convention adopted by the developer that should be respected and followed during the implementation. In the following some examples of URIs that targets different network entities :

/API/v1.0/Caches
/API/v1.0/Caches/Cache.id
/API/v1.0/Ports
/API/v1.0/Caches/Cache.id/Ports/Port.id

All the URIs starts with '/API/v1.0' which is the root of the URIs hierarchy, the version of the API is used since the API conforms to the agility principle, it will be patched and modified so we need versioning to recognize each version and also to address compatibility issues.

The first route corresponds to the entire set of caches in the network, using the identifier of a cache allows to interact with a specific cache in this set. In the same way we can get the entire set of ports in the network. The last route is related to a slice operation, this route identifies a port node that is attached to a Cache (here the Cache is defined as a slice). In general the URIs could have the following structures:

/API/{v1.0}/{node.id}
/API/{v1.0}/{slice.id}/{node.id}

1.4. Operations on resources:

The URI of the resource is used to reach it and do some actions e.g., create/update or delete the resource. HTTP verbs are used for this purpose, we use the output interface of the API to retrieve information about a node this interface is implemented using the GET method.

In the other hand the input interface of the API is modelled with create/update and delete operations implemented with the POST/PUT and DELETE methods.

➢ ***HTTP verbs (RFC 2616 Fielding):***

- **GET :** The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.
- **POST :** The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line.
- **PUT :** The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server. If the Request-URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI.
- **DELETE :** The DELETE method requests that the origin server delete the resource identified by the Request-URI. This method MAY be overridden by human intervention (or other means) on the origin server. The client cannot be guaranteed that the operation has been carried out, even if the status code returned from the origin server indicates that the action has been completed successfully.

➢ ***HTTP codes:***

Here we describe the most used HTTP codes :

- **200 OK :** Standard response for successful HTTP requests. The actual response will depend on the request method used. In a GET request, the response will contain an entity corresponding to the requested resource. In a POST request, the response will contain an entity describing or containing the result of the action.
- **201 Created :** The request has been fulfilled, resulting in the creation of a new resource
- **202 Accepted :** The request has been accepted for processing, but the processing has not been completed. The request might or might not be eventually acted upon, and may be disallowed when processing occurs.
- **400 Bad Request :** The server cannot or will not process the request due to an apparent client error (e.g., malformed request syntax, too large size, invalid request message framing, or deceptive request routing).
- **403 Forbidden :** The request was a valid request, but the server is refusing to respond to it. The user might be logged in but does not have the necessary permissions for the resource.
- **404 Not Found :** The requested resource could not be found but may be available in the future. Subsequent requests by the client are permissible.
- **405 Method Not Allowed :** A request method is not supported for the requested resource; for example, a GET request on a form which requires data to be presented via POST, or a PUT request on a read-only resource.

- **500 Internal Server Error:** A generic error message, given when an unexpected condition was encountered and no more specific message is suitable.

Giving the following URIs:

/API/v1.0/Router
/API/v1.0/Router/Router.id

Using the GET method with the first URI we will obtain all the information about all routers in the network. We can add a new router to this set using the POST method. For the second URI we can get information about a specific router identified with its ID using the GET method, or we can update it using the PUT method. Depending on the implementation of the function, DELETE is generally used to delete a node.

More complex methods and operations were implemented within our API to provide full functionalities to exploit the API for distributing content.

Chapter 4: The Implementation

1. API Implementation

In this section, we will first illustrate how the API will be used by presenting a content distribution use case, then see the different API building blocks that were implemented and integrated in the API.

In this section, we discuss the implementation that has been done to realize the API, we take as example the following generic network scheme that is composed of two caches, two file servers, different routers, a SDN controller and an HTTP Proxy.

1.1. Content distribution use case

The following generic network scheme is composed of a file server, cache, proxy, multiple switches, different clients consuming contents and the network is managed by a SDN controller and implements the API. As depicted in the figure 2, the clients will start consuming content from the file server, then depending on the popularity of requested contents, some will be cached in the cache unit, then all requests to these contents will be redirected to the cache using the proxy.

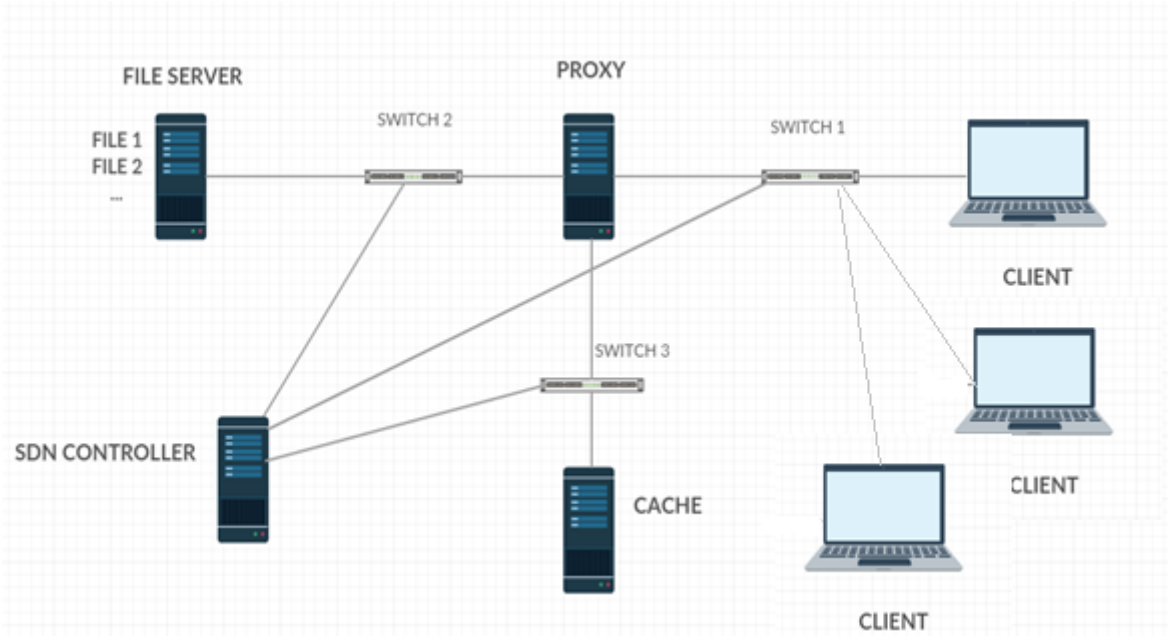

Figure 2 : Architecture adopted for content distribution scenario

1.2. Technical challenges

Different challenges were faced during the implementation of the content distribution API, the main ones are presented in this section and the solutions adopted are detailed in the coming sections.

- **Challenge 1 :** We are in a centralized environment, we need to find a way to reduce the load on the centralized entity in order to scale.

- **Challenge 2 :** Distribution of content demand over the different servers/caches, since depending on the location of the requested content the requests should be tweaked to reach the correct destination.

- **Challenge 3 :** HTTP flows are carried by TCP, to have information on the content requested by the client, we had to find a way to handle the TCP three-way handshake.

1.3. HTTP proxy to cope with the TCP three-way handshake issue

The first packets seen by the network does not have information about the content requested by the client, these packets are the TCP connection establishment packets or what we call the TCP three-way handshake packets.

To allow the network to determine which content is targeted by the client so that it can redirect the flows to the right destination (Server or cache) we had to come up with a solution that handles the TCP three-way handshake packets. The solution is to implement some HTTP proxies in the edge of the network so that the clients first establish a connection with the proxy then the proxy starts seeing the HTTP flows that carry information about contents.

Once the proxy knows which content is targeted by the client it will look for its location and then apply a packet tagging operation to distribute the requests to the correct destination.

1.4. Packet tagging technique

As mentioned before, the proxy uses a packet tagging technique once it determines the content requested by the client and its location. In our case, we are using IP source packet tagging, this technique is adopted due to the limitation of the proxy we are using. Cherryproxy does not use the API socket, it uses HTTPlib module that allows manipulation of the IP source and the port number of the packets. Since we are in a SDN context, we can modify IP addresses without impacting the behaviour of the network because the controller has always a global view on the network.

Packet tagging is used to distribute demands over the different servers/caches, of course there are other techniques that could be deployed, a naive one would be to install an Openflow rule for each content in the network. Packet tagging technique guarantees the stability of the network core since the tags are predefined, hence, manipulating new flows will not cause permanent flow table modification in the network.

1.5. Control-Plane states

The centralized approach adopted implies concentrating all information and decisions in a single point, this will cause high signaling load on the centralized entity thus impairing network performances, that is why we introduced the notion of control plane states that store, locally on the network components, the result of computations of the controller. This way, instead of querying the controller to know what operation to perform, network components query their local control plane states.

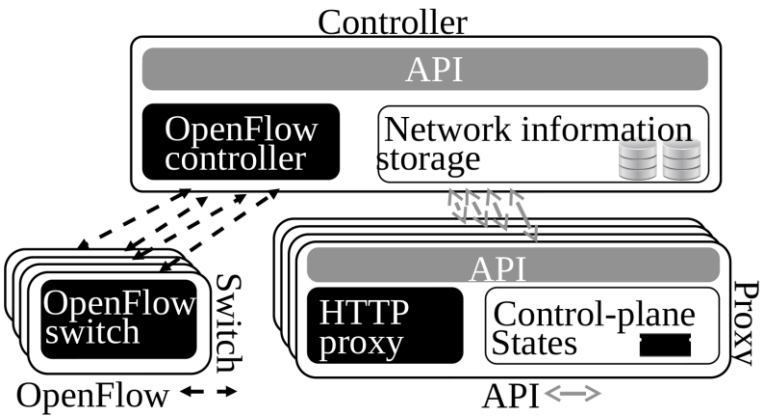

Figure 3 : SDN-enabled Architecture using control-plane states.

The main benefit behind using control-plane states is to reduce the load on the centralized controller, since each time a network component sees a new flow it will interact with the controller to know what operation it should perform. Using these control-plane states allow us also to adopt different architecture scenarios such as : fully-centralized, semi-centralized, and fully-decentralized. The first scenario is the case where all information and decisions are treated by the controller, i.e., no cache. The semi-centralized scenario is the situations where the control plane states only store the data plane decision made by the controller. Finally, the fully-decentralized control plane scenario is the one where all information are cached in the data plane network components.

1.6. Implementation steps

The first step was to define and store information of the different entities in the database, so we had to connect to our postgreSQL server through Flask then defining the different models (e.g., Cache model, Server model…). The database schema is the following:

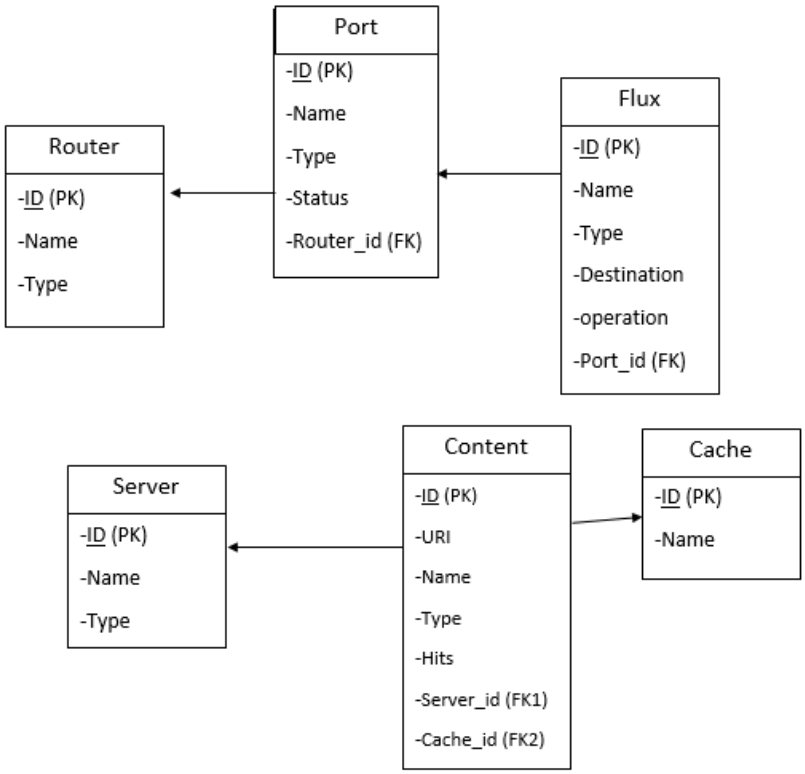

Figure 4 : Database relational schema

Below is an example of how the models are defined in our implementation:

```
class Router(db.Model):

    __tablename__ = 'Router'
    id = db.Column(db.Integer, Sequence('Router_id_seq', start=1, increment=1),
        primary_key=True)
    name = db.Column(db.String(20))
    type = db.Column(db.String(20))
    ports = db.relationship('ports', backref ='Belong_to', lazy='dynamic')

class ports(db.Model):

    __tablename__ = 'ports'
    id = db.Column(db.Integer, Sequence('ports_id_seq', start=1, increment=1),
        primary_key=True)
    name = db.Column(db.String(20))
    type = db.Column(db.String(20))
    status = db.Column(db.String(20))
    Router_id = db.Column(db.Integer, db.ForeignKey('Router.id'))
    flux = db.relationship('flux' ,backref='Sent_from_', lazy='dynamic')
```

Figure 5 : Code listing: Database models definition

Each element of these two models is defined with an ID, name, type… some of these information is unique (e.g., ID) and others are shared among elements of the same model (e.g., type). Once our models defined we implemented in each entity the corresponding methods, for example in a cache we have methods to create the cache, manipulate, contents in the cache and fetch contents from the providers. We have implemented in each entity the subset of methods related to their functionality.

In Flask there is the notion of decorators that is used to register a view function for a given URI rule, we give the following route annotation example:

```
@app.route('/API/V1.0/uploads/', methods=['POST'])
def upload():
        if request.method == 'POST':
        file = request.files['file']
        if file:
                filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'],filename))
            return redirect(url_for('upload'))
```

Figure 6 : Code listing: Upload function definition.

Here we define the route '/API/V1.0/uploads' that is related to the upload function and the HTTP method used (POST). All the functions in our implementation are related to a route (URI) defined in the same fashion using the route decorator and the HTTP methods.

As decided before the data representation format used in our API is JSON, Flask has a function that converts data to a JSON representation called JSONIFY, this function is very helpful since in our implementation we use SQL queries and we need to convert their responses to JSON to be manipulated by the API, below is an example of JSONIFY use:

```
@app.route('/API/V1.0/controller/maxhits/', methods=['GET'])
def maxhits():
    sub1 = db.session.query(func.max(Content1.Hits).label('max_hit')).subquery
        ()
    contenu1 = db.session.query(Content1).join(sub1, sub1.c.max_hit ==
        Content1.Hits).first()
    return jsonify(contenu1.serialize())
```

Figure 7 : Code listing: Function definition.

Once we have defined the different database models and the basic operations of each network component, we had to implement the essential building blocks that ensure the functionalities of the content distribution service. The main building blocks implemented are the following:

- **Content Popularity and prediction mechanism:** this function computes the popularity of each content in the network based on the number of 'hits' of each one, then it predicts this popularity in the future to decide whether to put the content in the cache or not. The prediction is based on a naïve predictor[2] that computes an average of the past

[2] the algorithm implemented is simple, since we are presenting a proof of concept this proves that we can implement more complex algorithms to compute the popularity of contents and predict it

observations and gives more importance to the recent ones, the predictor uses EWMA with the following formula:

$$S_t = \alpha * Y_t + (1- \alpha) * S_{t-1}$$

Y_t is the number of requests at a time period t.

S_t is the value of the WMA at any time period t.

- **Caching decisions:** the computations done by the previous function are retrieved by this function to decide what content will be cached. In this operation we had also to take into consideration the problem of content oscillation in case we have two contents that have approximately the same prediction, if one is already in the cache, it may be removed and returned just after, and this will increase the cost of installation of files in the cache.

- **Contents copy:** Now that we computed the popularity of each content in the network and know which one should be cached, all we have to do is to copy the contents from the source to the destination using this function.
- **Information dissemination:** the caching operation is accomplished after numerous steps, once it is done, these changes should be known by other network components(e.g., Proxy) to take the right decisions. Depending on the architecture scenario adopted, these informations could be retrieved or pushed directly by the controller using the API.
- **Traffic generator:** we implemented also a traffic generator based on a zipf's distribution since it has been proved that content requests in the internet follow a zipf-like distribution [23] and a poisson process to have independent requests. This generator could be used to simulate real traffic or tweak network behaviour.

The different functions were implemented with 1000 line of code, which confirms that we have made the right technological choices (REST, Flask, Cherryproxy...).

Chapter 5: The Evaluation

1. Evaluation

This chapter is dedicated to the definition of the evaluation setup, testing its network speed performance and finally presenting the experimentations conducted and the results.

1.1. Evaluation setup

Now that the API is implemented we have to test it, for that purpose, we adopt the setup depicted in figure 7. The setup is composed of different VMs each one hosting a running process (e.g., file server, Proxy …), an OpenVswitch virtual switch and a SDN controller are hosted by the physical host to bring connectivity and control to the network.

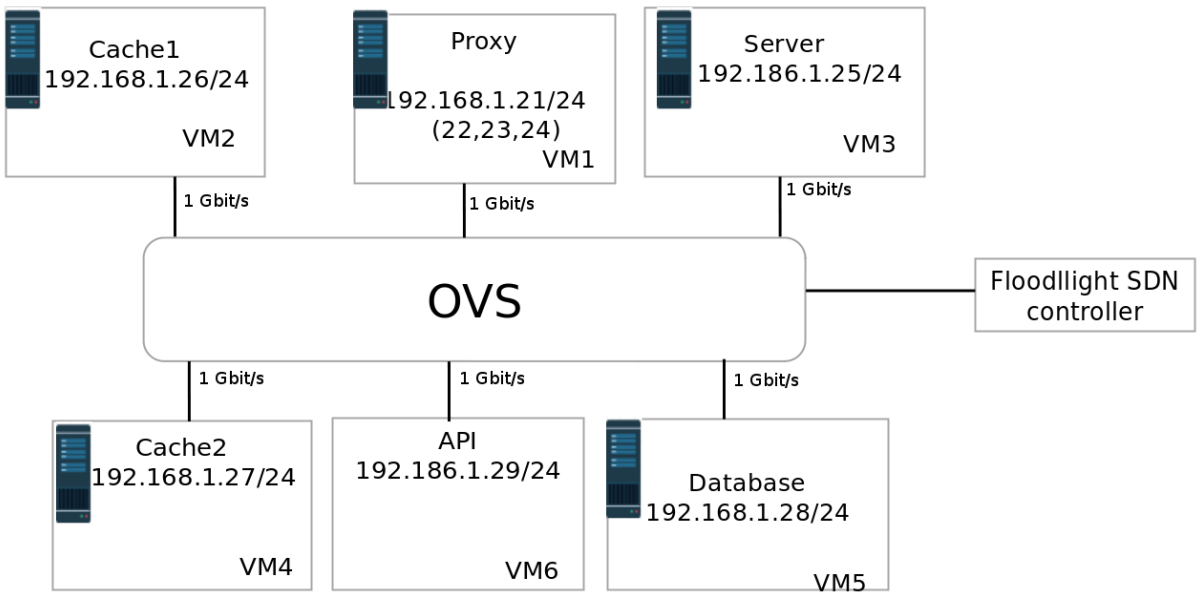

Figure 8 : Evaluation setup for the content distribution solution.

1.2. Evaluation setup test

The goal of testing our evaluation setup is to know the key performances that it delivers to the running processes connected through the network in order to be aware of the network capacity and limitation, in this way, we can predict the behaviour of the network. In the setup presented previously each host interface suppose to be capable of handling 1 Gb/s traffic, so we need to make sure that each link in the network we are emulating can reach the 1 Gb/s.

The Iperf tool is used to test the bandwidth available between each two machines when they talk alone in the network, and the case when all the machines are talking at the same time to see the difference in the performance and identify the characteristics of the emulated network.

- ✓ Iperf [24] is a network testing tool, it is capable of generating data streams be it TCP or UDP, as well as measuring the bandwidth of the network where we generate these streams. It is an active measurement tool. Iperf generates packets at any rate we choose.

- **Case 1 . Two machines communicating alone in the network :**

In this case all the machines were tested two by two and we got the following results:

Table 1 : Iperf bi-directional results between all the setup machines.

Machines	Server	Cache	Proxy	API/Controller	Database	Host
Server	22.4 Gbits/s	998 Mbits/s	996 Mbits/s	993 Mbits/s	995 Mbits/s	994 Mbits/s
Caches	998 Mbits/s	20.6 Gbits/s	999 Mbits/s	931 Mbits/s	992 Mbits/s	998 Mbits/s
Proxy	996 Mbits/s	999 Mbits/s	19.5 Gbits/s	991 Mbits/s	999 Mbits/s	964 Mbits/s
API/Controller	993 Mbits/s	931 Mbits/s	991 Mbits/s	21.2 Gbits/s	992 Mbits/s	954 Mbits/s
Database	995 Mbits/s	992 Mbits/s	992 Mbits/s	992 Mbits/s	22.5 Gbits/s	1 Gbits/s
Host	994 Mbits/s	998 Mbits/s	964 Mbits/s	954 Mbits/s	1 Gbits/s	45 Gbits/s

We ran bi-directional tests between all the machines, we expected to have the similar bandwidth values (1 Gbit/s) in both ways between each two different machines and higher rates when the tests are done within the same machine (i.e.,locally) since we don't use the ports limited to 1 Gbit/s. The table comes to confirm our expectations, it represents a symmetric matrix ($a_{ij} = a_{ji}$) , where all elements on the diagonal are more or less equal to 20 Gbits/s except the last one which is equal to 45 Gbits/s. The tests related to the results in red were done in a virtualized environment, this explains why there is a bandwidth reduction compared to the test done in the physical machine (result in blue) where the cost of virtualization is null.

These results confirm that each link in the network is capable of supporting 1 Gbits/s of traffic which means that we are capable of emulating a network where each link has a bandwidth of 1 Gbits/s.

- **Case 2 . All the machines communicating at the same time:**

Now we want to make sure that all the links can be saturated while all the machines are communicating at the same time. That is why, we connected a client to all the machines and launched bi-directional Iperf tests at the same time and we got the following results:

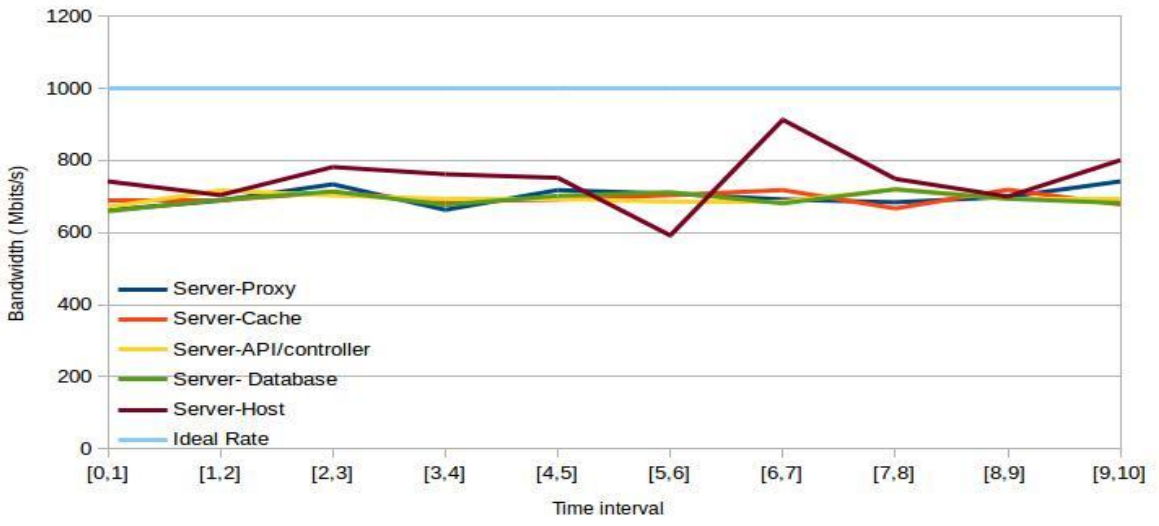

Figure 9 : Bandwidth utilization for each link in the setup with iperf tests running

The graph shows us that the bandwidth utilization of each link is around 800 Mbits/s, we can see the fairness of the hypervisor in term of process scheduling (i.e., Fairness in resource allocation for each VM) guarantees that no link is privileged on other links. We can also notice that the link between the Server and Host (physical machine) has slightly better performance than other links, since packets go through the hypervisor only once meaning that the cost of virtualization is lower than in the other cases. The links could not reach the same bandwidth values as before, and this is normal since all the machines are communicating at the same time, but we can see that all the links are fully utilized and can reach rates not too far from 1 Gbits/s (i.e., 750 Mbits/s).

✓ **Conclusion**

The tests shown before confirm that our evaluation setup is capable of emulating a 1 Gbits/s network, thus the evaluation results of our solution will not be biased by our setup.

1.3. Evaluation scenarios

We want to show the impact of our solution and the use of control plane states on the network load. To that aim, we adopt three different scenarios in our evaluation:

- **Scenario 1 : Fully-centralized**

This scenario is the case where all information and decisions are treated by the control (i.e., The notion of control-plane states is not used), each network component needs to interact with the control in order to take a decision or push new information that is important for future decisions.

- **Scenario 2 : Semi-centralized**

The semi-centralized or Hybrid scenario is the the situation where the control-plane states store only information related to routing decisions made by the controller, this way, the network component that need to take decisions will be able to do so very quickly (fetch information locally) and partially reduce the load on the centralized entity.

- **Scenario 3 : Fully-Decentralized**

In this case, all information will be written and read locally using the control-plane states, the network components will not have to interact with the controller to make decisions or push new information, periodically the controller will update the network state by fetching informations directly from the control-plane states of each network component (e.g., Proxy).

1.4. Results

In our evaluation we ran 25 experiments for each scenario, the experiments are done as follows:

- Generate traffic using a traffic generator to emulate content consumption.
- The controller updates its network state using the API to fetch new information (depends on the scenario adopted).
- The controller computes content popularity and predict it in the future.
- Based on the computations done before the controller takes caching decisions, then copies the most popular contents in the cache.

As mentioned before, we are using a traffic generator based on a poisson process and zipf distribution, in each scenario we evaluate the network performance at different content demand rates, the parameter used for that is the rate of the poisson process λ that we vary in [20;50;100;200;500;1000]. We expected to have some linear relation between the defined demand rates represented by the poisson rate and the corresponding packets generated, but we had the following result:

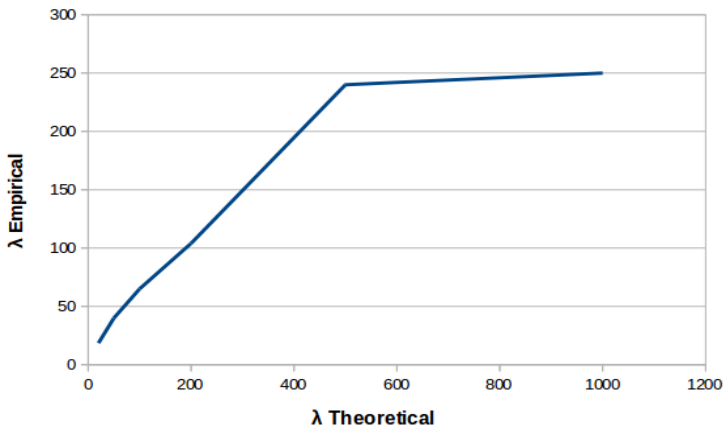

Figure 10 : The evolution of the empirical rate compared to the defined demand rates

The graph shows that there is not a linear relation between the defined rate of the poisson process and the rate of packets generated in the network and that there is a saturation after λ = 500, which means that we can not reach the desired data plane rates.

- The results of this evaluation are presented in the following graphs :

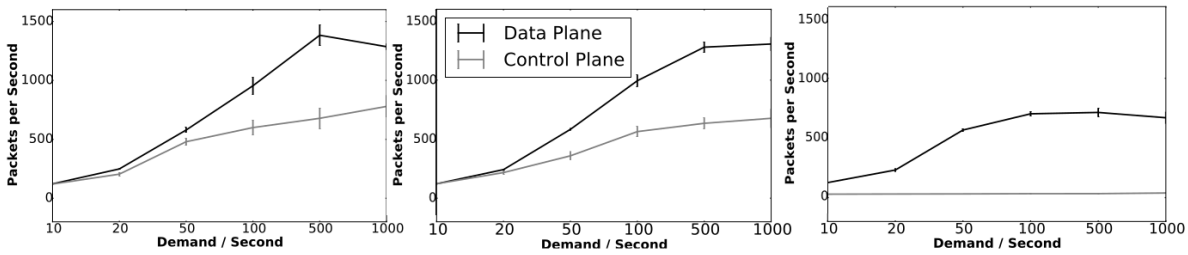

Figure 11 : Evolution of the control and data plane rates with content demand

These graphs represent the evolution of the data plane and the control plane with the content demand rate. The black curve represents the data plane and the grey curve represents the control plane, on the X-axis we have the content demand rate and on the Y-axis we have the rate of the generated packets (PPS - Packet per second). As said before, these graphs were obtained after running 25 experiments for each scenario and you can see in the graphs the confidence interval of 95% in each point that corresponds to the range of values we obtained during the evaluation.

The graph (a) and (b) show that there is a linear correlation between the control plane and the data plane rates, which means that as long as the content demand rate increases the control plane rate and the load on the controller will automatically increase. On the other hand, in the fully-decentralized scenario (c) we can see that the control plane rate is very low and stable, it does not depend on the content demand rate and this is guaranteed by the use of the control plane states.

We can notice also that in the centralized and hybrid scenarios we can reach almost the same data plane rates, but in the decentralized scenario we could not reach the same rates as before since the proxy needs more CPU cycles to process each flow and write locally the corresponding information. Reducing the load on the controller means using the control plane states which requires more CPU to obtain better data plane rates, in our case there is a trade-off between reducing the load on the controller and having good data plane rates.

Conclusion

In this work, we have proposed and evaluated a proof of concept to implement content distribution with Openflow, this approach enables the end user to leverage the flexibility a traditional SDN provides coupled with the content management properties we implemented. The proposed solution is based on an API implementation and a centralized architecture, it is transparent from the point of view of the server and client, and can be inserted in between with no modification at either end. We described our implementation choices as well as the overall architecture specification and we evaluated the performance in our evaluation setup. The solution was implemented in a small scale testbed, and demonstrated that it is capable of performing content distribution and network management. The results of our evaluation showed that using control-plane states guarantees a control plane load independent of the data plane rate.

Some immediate questions point to directions for future work: how can we optimize our system to obtain better network performance ? what is the gain of migrating the solution towards a powerful and dedicated platform ? Does control plane states still impact the data plane rates ? these questions will be handled as a natural extension to this work.

Bibliography

[1] Cisco, I. (2012). Cisco visual networking index: Forecast and methodology, 2011-2016. *CISCO White paper*, 2011-2016.

[2] George Xylomenos, Christopher N. Ververidis, Vasilios A. Siris, Nikos Fotiou, Christos Tsilopoulos, Xenofon Vasilakos, Konstantinos V. Katsaros, and George C. Polyzos "A Survey of Information-Centric Networking Research". *IEEE Communications Surveys & Tutorials Volume*: 16, Issue: 2, Second Quarter 2014.

[3] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. 2009. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies* (CoNEXT '09). ACM, New York, NY, USA, 1-12

[4] Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks".Communications Surveys and Tutorials, IEEE Com-munications Society, Institute of Electrical and Electronics Engineers, 2014.

[5] Https://en.wikipedia.org/wiki/Application_programming_interface

[6] Http://anr-disco.ens-lyon.fr/index.php?n=Main.INRIA

[7] Seyed Kaveh Fayazbakhsh, Yin Lin, Amin Tootoonchian, Ali Ghodsi, Teemu Koponen, Bruce Maggs, K.C. Ng, Vyas Sekar, and Scott Shenker. 2013. Less pain, most of the gain: incrementally deployable ICN. *SIGCOMM Comput. Commun. Rev.* 43, 4 (August 2013).

[8] Jennifer Rexford, Constantine Dovrolis " Future Internet Architecture: Clean-Slate Versus Evolutionary Research".Communications of the ACM, Vol. 53 No. 9, Pages 36-40.2010.

[9] Abhishek Chanda, Cedric Westphal, Dept of Computer Engineering, University of California " ContentFlow: Mapping Content to Flows in Software Defined Networks "

[10] Yoshio Sakurauchi, Rick Mcgeer, Hideyuki Takada "OpenWeb: Seamless Proxy Interconnection at the Switching Layer".International Journal of Networking and Computing.Vol 1, No 2 (2011).

[11] Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures Dissertation in partial satisfaction of the requirements for the degree of doctor of philosophy.

Webography

[12] Http://www.w3.org/TR/soap/

[13] Https://tools.ietf.org/html/rfc7159

[14] Https://www.w3.org/XML/

[15] Http://flask.pocoo.org/

[16] Https://www.djangoproject.com/

[17] Https://www.opendaylight.org/

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

[19] Http://floodlight.atlassian.net/

[20] Https://www.postgresql.org/docs/9.4/static/release-9-4.html

[21] Https://www.mongodb.com/

[22] Https://fr.wikipedia.org/wiki/NoSQL

[23] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, Scott Shenker " Web Caching and Zipf-like Distributions: Evidence and Implications".IEEE INFOCOM. 1999

[24] Https://iperf.fr/iperf-doc.php

[25] Http://www.sqlalchemy.org/

[26] Http://initd.org/psycopg/