CONTENT DELIVERY IN SOFTWARE DEFINED NETWORKS

BY ABHISHEK CHANDA

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Dipankar Raychaudhuri

and approved by

_____

_____

_____

New Brunswick, New Jersey

May, 2013

ABSTRACT OF THE THESIS

Content Delivery in Software Defined Networks

by Abhishek Chanda

Thesis Director: Professor Dipankar Raychaudhuri

Information Centric Architectures view content as the narrow waist of the networking stack. This abstraction allows routing based on the content name, rather than the network locator of the content consumer and producer. We present *ContentFlow*, an Information Centric network architecture which supports content routing by mapping the content name to a OpenFlow defined flow based on TCP and IP semantics. And, thus enables the use of OpenFlow switches to achieve content routing over a legacy IP architecture. *ContentFlow* is viewed as an evolutionary step between the current IP networking architecture, and a full edged ICN architecture. It supports content management, content caching and content routing at the network layer, while using a legacy OpenFlow infrastructure and a modified controller. By efficiently using the content information available in the network, *ContentFlow* supports efficient traffic engineering. Also, *ContentFlow* is transparent from the point of view of the client and the server, and can be inserted in between without modification at either end. The architecture and implementation of *ContentFlow* on top of the existing OpenFlow software defined networking framework is described. Performance of *ContentFlow* is evaluated using a prototype implementation of an enterprise SDN network with Floodlight controller and multiple virtualized OpenFlow switches. The results show that *ContentFlow* does result in reduced content access delay in comparison to a legacy architectures.

Acknowledgements

I would like to take this opportunity to thank my adviser Prof. Dipankar Raychaudhuri for his guidance, motivation and constant encouragement. I have always been inspired by the simplicity of his logic, his acute engineering insights and the relevance of his advice - technical and otherwise.

I would also like to thank Dr. Cedric Westphal who was my mentor at Huawei for his guidance throughout my internship. And also, Prof. Yanyong Zhang and Prof. Janne Lindqvist for their constant support during my time here at WINLAB. Finally, I am grateful to Ivan Seskar who has been patient with all of my queries, however trivial those were. Also, I would like to thank Prof. Marco Gruteser for being a part of my thesis committee.

I would also like to thank the WINLAB staff for all the facilities and help in academic matters. I am grateful to all my friends here at WINLAB and at Rutgers, without whom this experience would not have been the same. In particular I would like to thank Nileema, Atisha, Gautam, Sam, Shweta, Shraddha, Shreyosee, Sowrabh, Akash, Tam, Anukriti, Murtuza, Dhara, Prateek and specially, Dr. Ankita Sen.

Table of Contents

List of Figures

Chapter 1

Introduction

The architecture of the current Internet was conceived in the 1960 and 70's with a set of assumptions, the primary assumption being that hosts are connected by point to point wired links and that hosts do not move (and change their routable address) often. On the basis of these assumptions, transport protocols like TCP and UDP were designed. In the last 50 years, computing equipment has become cheap and ubiquitous owing to Moore's law [1]. Moreover, these devices have different forms of Internet connectivity and sensors that can generate content. The proliferation of these mobile devices has broken the basic assumptions of the Internet in a number of ways:

- Being mobile, these devices often change location and hence their network attachment point, which results in frequent changes to their IP address. This effectively breaks the assumption of having static devices which do not change their routable address often.

- These devices often have multiple network interfaces, often with different technologies having different routable addresses. This breaks the assumption that a host will have only one routable address.

- Often, a web service needs to communicate with a number of devices selected by some constraint. An example of such a scenario is that of push notifications where a web service wants to send a notification to a group of devices. This pattern breaks the assumption of having point to point links between communicating devices.

Over the years, a number of solutions to these problems have been proposed that conforms to the design of the existing Internet. This approach promotes incremental

deployment of patches to the existing protocol suite. We will describe some of these in the related works section.

1.1 Related works

Many new Information-Centric Network (ICN) architectures have been proposed to enable next generation networks to route based upon content names [2, 3, 4, 5]. The purpose is to take advantage of the multiple copies of a piece of content that are distributed throughout the network, and to dissociate the content from its network location. However, the forwarding elements in the network still rely on location (namely IP or MAC addresses) to identify the flows within the network.

One issue is that content is not isomorphic to flows, especially when it comes to network policy. Consider as an example two requests from a specific host to the same server, one for a video stream and one for an image object. Both these requests could be made on the same port, say port 80 for HTTP as is most of the Internet traffic. From the point of view of a switch, without Deep Packet Inspection (DPI), they will look identical. However, the requirements for the network to deliver proper service to these flows are very different, one with more stringent latency constraints than the other. Flows are not the proper granularity to differentiate traffic or to provide different sets of resource for both pieces of content.

Yet, flows are a convenient handle as they are what switches see. Consider the recent popularity of Software-Defined Networking (SDN) and OpenFlow [6] in particular. This framework introduces a logically centralized software controller to manage the flow tables of the switches within a network: by using the flow primitive to configure the network, a controller can achieve significant benefits. A mechanism is needed to bridge the gap between the flow view as used in SDN and achieving policy and management at the content level, as envisioned by ICNs.

A separate body of work exists which discusses traffic engineering in Internet scale architectures.The problem of traffic engineering has been studied extensively since the early days of the Internet and telecom networks. RFC 3272 provides an overview of TE

in the Internet [7] and also defines traffic engineering in this context. Reference [8] is another work which describes TE in an IP network and shows that traditional shortest path routing protocols can be used for TE in IP networks by instrumenting link weights based on traffic.

More recently, with the advent of cloud computing, a lot of data intensive applications have moved to data centers. Thus, traffic engineering has found applications in data center networking with its own set of challenges. A number of authors have identified the merits of traffic engineering of *elephant flows* [9, 10] – flows that carry a large amount of data – in a data center network. Curtis et. al. proposed Mahout [9] which aims to identify elephant flows by installing a shim layer on end hosts. The layer monitors TCP send buffer sizes on the hosts and reports them back back to an OpenFlow controller. The controller can aggregate all statistics collected from a path to identify elephant flows on that path. Those flows can then be re-allocated according to some optimization criteria. One major issue with this architecture is that it requires modification of end hosts, which is not always desirable. In this work, we keep track of content in a similar manner as Mahout keeps track of elephant flows in the data center. [10] proposed Hedera on similar grounds which works by instrumenting end switches in a data center network. None of these works focus on ICN and thus fail to take advantage of the extra information associated with knowledge of content properties above the transport layer. For instance, identifying elephant content in an ICN does not require any end point support.

Very recently, some authors have started to investigate TE in an ICN. One such work is [11] which proposed Content Aware Traffic Engineering (CaTE). This works by exploiting path diversity content information in a ISP network. Xie et. al. proposed a method to decouple traffic engineering and collaborative caching in a network [12]. Chanda et. al. proposed the inclusion of non-forwarding elements which are essential in an ICN under a centralized control layer ([13, 14, 15]). Our present work is a natural extension of this work.

Chapter 2

Problem formulation

We propose the use of OpenFlow based flows to handle content in a network. This will allow *ContentFlow* to operate on existing IP architectures and yet provide a transparent ICN type forwarding paradigm. There are two challenges to this approach:

- OpenFlow architecture does not support including non-forwarding elements in a network which we need to support ICN functionality.It is the common view of most future Internet architectures [2], [3], [4] and many commercial products [5] combine a switching element with additional capability, such as storage and processing; these capabilities need to be advertised to the control plane so that the programmable policy takes them into account.

- OpenFlow does not support differentiated treatment of traffic at any finer granularity than that of a flow. Currently, it OpenFlow ignores the specic content and provides a forwarding behavior that is per-ow based, where the ow is dened as a combination of the source and destination MAC and IP addresses and protocol ports plus some other elds such as MPLS tags, VLAN tags; however, we can consider a specic content le: while the source and destination and protocols of two les might be identical (say, from the same host to the same data center using TCP port 80), the two les might require different forwarding treatment (say, one is a streaming video with some delay constraints while the other is a picture with less strict delay requirements). Many architectures attempt to place content as the focus of the architecture [6], [7], [8], [9] and this is not supported by the current SDN proposals.

In this rst step towards a more general denition of SDN, we focus on a particular use case. Namely, we will add storage primitives to SDN as an example of a new element that can be under the controllers reach. Other processing elements could be included as well. And we will add content handling mechanisms to enable seamless switching based on content. To demonstrate the feasibility of our approach, we have implemented a content management framework on top of a SDN controller and switches which includes a distributed, transparent caching mechanism

2.1 Storage primitives

In-network storage is expected to be deployed in most routers in many future network architectures. In a SDN architecture with an abstracted view of the network being held at a controller, in-network storage elements (storage enabled routers, cache etc.) need to be able to advertise to the controller their ability to store content. And since the controller has a high level view of the whole network, it can use the storage elements based upon its need and its network performance targets. There are two types of in network storage: 1) Forwarding elements in the network can have storage associated with them. In a rst step, storage in a routing element should be understood as a store-and forward capacity, rather than a distributed le system: namely a le can be stored in the network for trafc engineering purpose or for caching. This is for administrative reasons: we assume that the network operators own its network elements and use their resource to optimize their trafc delivery. Later on, storage can be extended to include repositories for the output of processing tasks, assuming a framework is in place to allocate and virtualize storage to a variety of entities. 2) A network can have non-forwarding elements which can store content at different levels. Again, these elements do not form a distributed le system, rather they are a part of a distributed caching system which is synchronized over some caching protocol. Currently, the most commonly used caching protocols are ICP and HTCP. The current design of the Internet caching protocols do not allow virtualization of the storage elements. In this work, we propose a method to virtualize in network storage at the controller. Upon adding a storage element in the network, the following conguration needs to take place:

- The storage element needs to identify the location of the controller
- The storage element needs to set up a session with the controller as any other switching element
- The storage element needs to advertise its capability to the controller, in a way that the controller can understand
- The controller needs to integrate the capability of the storage element in its control policy
- The storage element needs to be able to provide some usage statistics to the controller, either at periodic interval or upon request from the controller
- The storage element needs to be able to refresh the association with the controller by maintaining some keep alive mechanism; and to be able to tear down the session when it terminates

It can be observed that this list of functionality is very similar to what a forwarding element does when it is deployed in a network. Thus, we hypothesize that extending the same (or similar) set of functionality to a non-forwarding element would be possible. Essentially, we need to expand the OpenFlow protocol with some new elds to support the tasks mentioned.

2.2 Content primitives

The next important step after establishing a set of storage primitives in a SDN is to establish a set of content primitives. This means that the content needs to be properly managed and identied throughout the network. We focus here on content described by HTTP requests, as it constitutes a large part of the Internet trafc. Expanding these principles described here to other protocols and le transfer mechanisms will then be achieved through simple protocol parsing in a network element.

Content based routing (and forwarding) dictates that routing should be performed based on content, and not on the source and destination address obtained by the DNS

resolution (unless the DNS mapping is updated dynamically and in real time). The benets of this approach are straightforward; popular content gets higher priority and is probably cached somewhere close to where it is popular, resulting in reduced access latency. To see this in action, assume that a host would like to access le A and le B, which are both located on server S. However, le A happens to be very popular, and le B is not popular at all. This means that a copy of le A has been most likely replicated in some caches, while le B is exclusively found at its publishers location, server S. Routing based upon the typical IP-based match lters would route both HTTP requests for le A and for le B to server S. Routing on content on the other hand would let the request for le B go to server S, and the request for le A would be send to a nearby cache which contains the requested object. And if the cache is located somewhere close to the clients who request the content, this would result in decreased latency and delay.

The following needs to be supported:

- Content identication: a network element must be able to identify content preferably close to the source. There can be a number of possible methods of doing this, content can be identied based on a combination of HTTP and TCP semantics (all trafc on port 80 on a webserver is content). Another option is to put a DPI module on the customer facing edge of the network. The third and the most robust option is to include special content matching elds in OpenFlow that forwarding elements can match against.

- Content routing: once content has been identied and the destination has been found, we would need to route the packets to a given destination. This can be implemented by augmenting an OpenFlow controller with a module that can push custom ows to forwarding elements.

- Integration with the cache mechanism: standard Internet caching systems are complicated. They expect a set of HTTP headers to congure caching behavior. It is a design decision whether an OpenFlow based solution comply with existing Internet caching mechanisms. For the sake of brevity, in this work we ignored integration with Internet caching.

Chapter 3

Description of ContentFlow

The major design philosophy here is to implement content based management using existing elements as much as possible. We present here the *ContentFlow* architecture, which enables allocation of content-dependent ow headers. We use a combination of the source, destination ports and IPv4 addresses, but not that we could also use IPv6 addresses and use the lower bits in the IPv6 address to embed content information. We take the point of view of a network operator, and assume that content is requested by an end-user from a server, both of which can be outside of the network. Thus, *ContentFlow* has been designed so that it can be placed as a plugin network between a service provider and a consumer network.

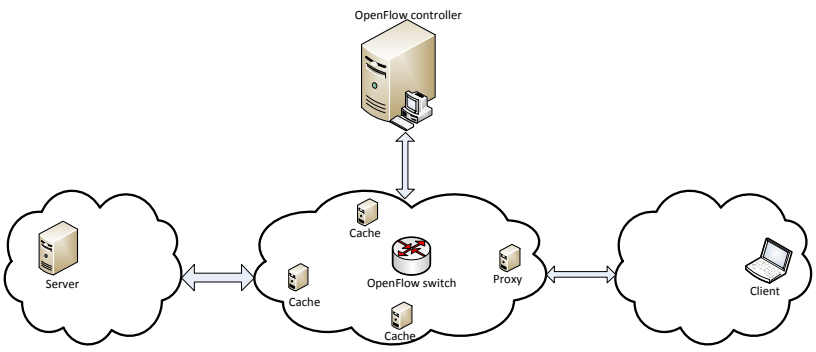

Figure 3.1: Network model

Fig 3.1 shows the placement of such a network. It is an OpenFlow network which is

managed by a controller (without loss of generality, we can assume only one controller for the network). The controller runs a content management layer (within the control plane) that manages content names, translates them to routable addresses and also manages caching policies and trafc engineering. The control plane translates the information from the content layer to ow rules which are pushed down to switches. Some of the switches in the network have the extra ability to parse content metadata and pass on to the content management layer in the controller.We assume that the service provider network connects to the caching network using one or more designated ingress switches.

The key design concern is to allow multiple ows from the same source/destination pair and to provide this handle to the different network elements. We consider both switches and caches to be network elements. Thus we need to identify a piece of content, assign it a ow identier for routing purpose, yet at the same time, provide a caching element with this ow identier and the original content name, so as to demultiplex content in the cache.

We perform a separation of content and its metadata as early as possible. Thus, we try to separate out the metadata for the content close to the source and put it in the control plane. In our implementation, the metadata consists of the le name parsed from the HTTP GET request and the TCP ow information. Thus, it is a five tuple of the form $\langle file\ name, destination\ IP, destination\ port, source\ IP,$
$source\ port\rangle$. A high level overview of the architecture is shown in figure 3.2.

One key aspect is that content routing in a SDN network requires to proxy all TCP connections at the ingress switch in the network. This is due to the TCP semantics: a session is rst established between a client and the server, before an HTTP GET is issued to request a specic content. Routing is thus performed before the content is requested, which prohibits content routing. On the other hand, content-routing requires late binding of the content with the location. Proxying TCP at the ingress switch ensures that no TCP session is established beyond the ingress switch. Only when the content is requested, then the proper route (and TCP session) can then be set-up.

Figure 2 shows the different architecture elements of ContentFlow: a proxy at the

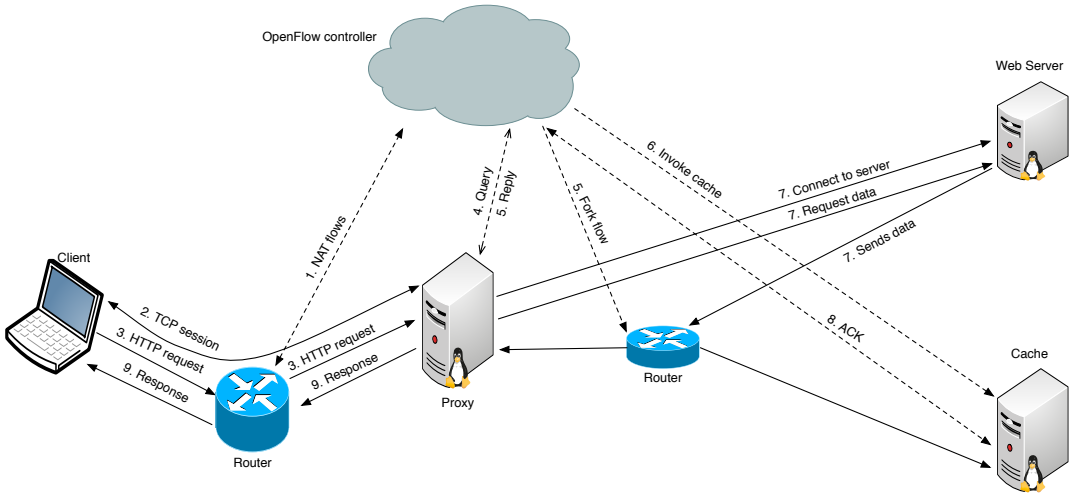

Figure 3.2: Overview of the architecture

ingress to terminate the TCP sessions and support late binding; OpenFlow-enabled switching elements; caching and storage elements which can transparently cache content; a content controller expanding an OpenFlow controller to include content features; and the protocols for the controller to communicate with both the caches and the proxies (the controller-to-switch protocol is out-of-the-box OpenFlow). The different elements and protocols are detailed later.

The system works as follows, the first case will be a cache miss where the requested content is not present in the cache. Thus, the request will be redirected to the original destination server. Once the content has been cached, subsequent requests will be redirected to the cache.

1. When a client (in the client network) tries to connect to a server (in the provider network), the packet reaches the ingress switch of the caching network. This switch does not find a matching flow and sends this packet to the controller.

2. The controller assigns a port number (on the proxy) to this client server pair.

3. The controller installs static rules into the switch directly connected to the client to forward all HTTP traffic from the client to the proxy (on the selected port) and vice-versa. The TCP proxy terminates all TCP connections from the client. At this point, the client sets up a TCP session with the proxy and believes that

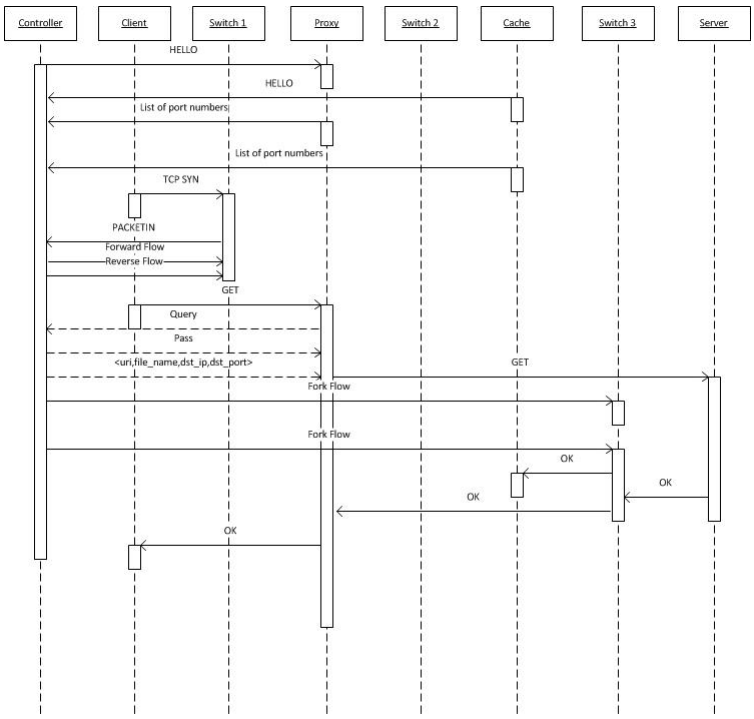

Figure 3.3: Sequence diagram of the system

it has a TCP session with the server.

4. The client issues an HTTP request which goes to the proxy.

5. The proxy parses the request to extract the name of the content and the web server to which the request was sent.

6. The proxy queries the controller with the file name (as a URI) asking if the file is cached somewhere in the network.

7. Since the content is not already cached, the controller returns "none". The proxy updates the controller with the metadata from HTTP request and directs the connection to the original web server.

8. The controller then determines whether the content from the web server should (or should not) be cached. To cache the content, it computes a forking point where the connection from the web server to the proxy should be forked so that a copy of the packets will be duplicated to the cache.

9. The controller installs a rule in the switch and invokes the cache. The controller notifies the cache of the content name and of the flow information to map the content stream to the proper content name. The controller also records the location of the cached content.

10. The cache saves the content with the file name obtained from the controller.

11. The other copy of the content goes back to the proxy and in the egress switch, it hits the reverse ow which re-writes it's source IP and port to that of the server.

For the cache hit case, steps 1 to 5 are the same and the controller returns a cache IP in step 5. The proxy connects to the cache which serves back the content. On it's way back, the content hits the reverse flow and the source information is re-written making the client believe that the content came from the original server. Figure 3.3 shows how the system works.

3.1 Proxy

The proxy is a transparent TCP proxy that is located in the caching network. In our setup, OpenFlow is used to make the proxy transparent by writing reactive flows to appropriate switches. The proxy is the primary device that is responsible for separating out content metadata and putting it on the control plane, thus it must intercept packets from the client and parse relevant information. The proxy communicates with the controller through REST API calls. Once the client has setup a connection with the proxy, it issues a HTTP request which the proxy intercepts. The proxy parses the request to extract content metadata and queries the controller. Algorithm 1 describes how the proxy works.

While booting up, the proxy starts an instance of itself on a bunch of ports. It also reads a configuration file to know the IP address and port of the OpenFlow controller to which it connects. The proxy then establishes a session with the controller and gives it a range of usable port numbers to be used as content handles. Now, when a client tries to connect to a server, the controller picks the first unused port number

and redirects the connection to that port. From this point, this port number acts as a handle for the content name. The controller maintains a global mapping of the form $\langle content\_name, client\_ip\_port, server\_ip\_port, handle \rangle$ which can be queried either using *content_name* or *handle* to retrieve the other parameters.

```
Listen on proxy port;
if a GET request arrives then
    Parse the file name from the request;
    Construct content name by combining the destination URI and the file name;
    Query controller with the content name;
    if the controller returns an IP address then
        Redirect all requests to that IP address;
    else
        Update controller with the file name;
        Pass the request unmodified;
    end
    Add the port to the list of free ports;
    Update the controller;
else
    Do not proxy;
end
```

Algorithm 1: Proxy Algorithm

3.2 Cache

In our design, when the cache receives a content stream to store, the cache will see only responses from the web server. The client's (more accurately, proxy's) side of the TCP session is not redirected to the cache. This scenario is not like generic HTTP caching systems like Squid which can see both the request and response and cache the response. In our design, we want to avoid the extra round trip delay of a cache miss, so we implemented a custom cache that can accept responses and store them against request metadata obtained from the controller. The cache implements algorithm 2.

We handle streaming video as a special case which is delivered over a HTTP 206 response indicating partial content. Often, in this case, the client will close the connection for each of the chunks. Thus, when the cache sees a HTTP 206 response, it parses the `Content-range` header to find out how much data is currently being transmitted and when that reaches the file size, it knows it has received the complete file and then

it can save it.

```
Listen on cache port;
Start webserver on cache directory;
if a HTTP response arrives then
    if The response is a 206 partial content then
        Extract the content-range header to know current range being sent;
        while Server has data to send do
            Append to a file
        end
        Query the controller with the source IP and port;
        if the controller sends back a file name then
            Save the response with the file name;
        else
            Discard the response;
        end
    else
        Lookup the source IP of the response;
        Query the controller with the source IP and port;
        if the controller sends back a file name then
            Save the response with the file name;
        else
            Discard the response;
        end
    end
else
    Serve back the file using the webserver;
end
```

Algorithm 2: Cache Algorithm

3.3 Controller

The controller can run on any device that can communicate with the switches and in our case, the non-forwarding elements (in most cases, it is placed in the same subnet where the switches are). It maintains two dictionaries that can be looked up in constant time. The *cacheDictionary* maps file names to cache IP where the file is stored, this acts as a global dictionary for all content in a network. *requestDictionary* maps destination server IP to file name, this is necessary to forward content mete data to the cache when it will save a content. The controller algorithm is described in 3.

Install static flows for NAT in the switch to which the client is connected;
cacheDictionary ← {}
requestDictionary ← {}
if *proxy queries with a file name* **then**
| Lookup cache IP from *cacheDictionary* and send back
end
else if *proxy sends content meta data* **then**
| Insert file name and destination IP to *requestDictionary*
| Compute the forking point for a flow from destination IP to proxy and destination IP to cache
| Push flows to all switches as necessary
| Invoke the cache and send it the file name from *requestDictionary*
| Insert the file name and cache IP in *cacheDictionary*
end

Algorithm 3: Controller Algorithm

3.4 Flow definitions

We will need three flows per communicating server client pair (other than all flows that are necessary to enable usual communication). Two flows to manage the connection between the client and the server in both directions and one flow to fork content. The flows are defined as follows, forward flow is the flow that enables the client to communicate with the proxy. It re-writes the destination IP, MAC and port of packets to that of the proxy, effectively making the OpenFlow switch a DNAT box (which does destination NAT by re-writing the destination MAC, IP and port of all outgoing packets) . Reverse flow enables communication from proxy to client. It re-writes the source IP, MAC and port to that of the original server, effectively making the switch act as a SNAT box (which does source NAT by re-writing the destination MAC, IP and port of all incoming packets) . The fork flow enables sending the content to two different boxes. Let X be the port number the controller selects for the particular server client pair.

Forward flow:

if src_ip = client_ip and dest_ip = server_ip then mod_dest_ip = proxy_ip and mod_dest_mac = proxy_mac and mod_dest_port=X

Reverse flow:

if src_ip = proxy_ip and dest_ip = client_ip then mod_src_ip = server_ip and mod_src_mac

= server_mac and mod_src_port = server_port

Fork flow:

if src_ip = server_ip then flood and mod_dest_ip = cache_ip and mod_dest_mac = cache_mac and mod_dest_port = cache_port and flood

Chapter 4

Content management layer and ContentFlow controller

As mentioned earlier, we propose an augmentation to a standard OpenFlow controller layer to include content management functionality. This layer handles the following functionality:

- **Content identification:** we propose content identification using HTTP semantics. This indicates, if a client in a network sends out a HTTP GET request to another device and receives a HTTP response, we will conclude that the initial request was a content request which was satisfied by the content carried over HTTP (however, the response might be an error. In that case we will ignore the request and the response). Further, we propose that this functionality should be handled in the proxy since it is directly responsible for connection management close to the client. The content management layer gathers content information from the proxy which parses HTTP header to identify content.

- **Content naming:** as with a number of content centric network proposals, we propose that content should be named using its location. Thus, if an image of the name *picture.jpg* resides in a server whose name is *www.server.com* in a directory called *pictures*, the full name for the specific content will be *www.server.com/pictures/picture.jpg*. Such a naming scheme has several advantages; 1) it is unique by definition, the server file system will not allow multiple objects of the same name to exist in a directory. Therefore, this naming scheme will allow us to identify content uniquely. 2) this scheme is easy to handle and parse using software since it is well structured. 3) this scheme is native to HTTP. Thus it allows us to handle HTTP content seamlessly. As mentioned before, in our implementation, the content name is derived by parsing HTTP header for

requests.

- **Manage content name to TCP/IP mapping:** to enable the content management mechanism to work on a base TCP/IP system, we need to map content semantics to TCP/IP semantics (end point information like port and IP) and back. The content management layer handles this by assigning a port number to a specific content; the data structure can be looked up using the port number and server address to identify the content. The OpenFlow flows can be written to relevant switches based on that information. The whole process is described in figure **??**. Note that, the number of (server address,port combinations) in the proxy is finite and might be depleted if there are too many requests. Thus, freed up port numbers should be reused and also, the network admin should use sufficient number of proxies so that the probability of collision is sufficiently small.

- **Manage Content Caching Policy:** the *cacheDictionary* can be expanded to achieve the desired caching policy, but taking all the distributed caching available into account. For instance, the controller can append to each content some popularity information gathered from the number of requests, and thus decide which content to cache where based upon user's request history.

It is easy to note here that, since the number of ports in a proxy is finite, there is a non zero probability that a large number of requests to a specific server might deplete the pool of possible (port,server) combinations.

- **Ensuring availability:** since there is a limit on the number of potential requests to a given server, we must make sure there are enough port numbers to accommodate all possible content in our network. Since contents map to a port/proxy/server combination, the number of concurrent connections to a server is as limited by the number of available source ports between a given proxy/server pair. Further, the number of proxies can be increased, by adding virtual proxies with different addresses, and thus increasing the flow space. IPv6 could be used there as well to increase the flow space size and demultiplex different content over different flows. The use of a flow filter for a restricted period of time can

be translated as a queuing theoretic issue and has been proposed in the case of network address translation for IPv4-IPv6 migration in [**?**]. The analysis therein translates directly into dimensioning the number of concurrent content requests that a proxy/server pair can perform.

- **Ensuring consistency:** the consistency constraint implies all clients should receive their requested content only and not each others. In our system, this translates to the constraint that once a port number is designated for a content, it should not be re-used while that content is live in the network. This is ensured using the controller. The controller maintains a pool of port numbers from which it assigns to content. Once a port number is in use, it is removed from the pool. When the client closes the connection to the proxy, the proxy sends a message to the controller and updates the list of free port numbers, which the controller puts back in the pool.

Thus while the proposed system ensures availability and consistency, it is not fully partition tolerant since the whole architecture is sensitive to the latency between the OpenFlow controller and the network elements. This limitation however, is from OpenFlow and not our architecture which leverages OpenFlow.

Chapter 5

Supporting Traffic Engineering in *ContentFlow*

Information Centric Architectures handle content by definition. Therefore, they always have a holistic view of the content at any point of time. Our Traffic Engineering scheme is motivated by the observation that given an ICN like *ContentFlow* , it is trivial to extract content information and use that to enable efficient traffic engineering. For this, we will need some extra functionality as described below:

5.1 Content metadata extraction

Since we intend to implement TE and firewalling by using content metadata, we will need a mechanism to extract that information. We can define two levels of abstraction in this context. The first one is strictly at the network layer, and takes advantage of the ICN semantics. The second goes into the application layer.

- *Network-layer mechanism to extract content length:* Since in an ICN, content is uniquely identifiable, the controller can recognize requests for new content (i.e. content for which the controller holds no metadata in the key-value store). For new content, the controller can set up a counter at a switch (say, the ingress switch) to count the content flow size[1]. The controller can also request that the flow be cached, and obtained the full object size from the cache memory footprint. When the content travels through the network later, a look-up to the key-value store allows to allocate resource accordingly. Further, the content that is observed for the first time can still be classified into *elephant* and *mice* flows on the fly based on some threshold and allocated accordingly to optimize some constraint.

[1]In this case, the flow size will include some header overhead that the controller can subtract later on.

- *Application-layer mechanism to read HTTP headers in the ingress switch:* In the previous case, content size is not available for content which is observed for the first time. However, this can be extracted by parsing the HTTP headers. This will allow the controller to do elephant flow detection and take appropriate actions early. Though it is difficult to implement, an advantage of this method is that it will allow us to do TE and firewalling from the first content occurrence.

5.2 Controller driven coordination

Once the network elements have the ability to extract content metadata, they will need to announce their ability to the controller. We propose that this should be done in-band using the OpenFlow protocol since it supports device registration and announcement features. This essentially involves the following steps

- Asynchronous presence announcement, where a device announces its presence by sending a hello message to the assigned controller.
- Synchronous feature query, where the controller acknowledges the device's announcement and asks it to advertise its features.
- Synchronous feature reply, when the device replies with a list of features.

After these steps, the controller has established sessions to all devices and knows their capabilities, it can then program devices as necessary.

Given the setup described, the controller can now have extra information about content in a network. Also, the SDN paradigm allows the controller to have a global view of the network. Thus, the platform can support implementation of a number of services. Here, we will discuss four services as example.

- *Metadata driven traffic engineering* Amongst various content metadata parameters, since the controller knows the content length, it can solve an optimization problem under a set of constraints to derive paths on which the content should be forwarded. Modern networks often have path diversity between two given devices. This property can be exploited to do traffic engineering. This approach

of doing traffic engineering is efficient and salable since it does not require the service providers to transfer content metadata separately, which saves network bandwidth at both ends.

- *Differentiated content handling* The DPI driven mechanism described earlier enables us to have rich content metadata. Thus, assuming such a content metadata extraction service, the content management layer in the controller can take forwarding decisions based on the MIME type of the content. A network administrator can describe a set of policies based on content types. Let us take delay bound for example. If the MIME type is that of a real time streaming content (a video clip), it can select a path which meets the delivery constraints (the delay bound which has been set). If none of the paths can satisfy the bound, the path with the lowest excess delay will be selected. This approach can be combined with the traffic engineering described above to handle multiple streaming content on a switch by selecting different paths for each.

- *Metadata driven content firewall* We can envision a metadata driven content firewall in the network. When a piece of content starts to enter the network, the controller knows how long is it. Thus, it should be possible to terminate all flows handling the content after the given amount of data has been exchanged. This mechanism acts like a firewall in the sense that it opens up the network only to transmit the required amount of data. We argue that this mechanism provides stronger security than traditional firewalls. With a traditional firewall, the network administrator can block a set of addresses (or some other parameters). But the attacker can always spoof IP addresses and bypass the firewall. However, with this proposed mechanism, the network will not let content from spoofed IP addresses pass through since it knows that the content has already been transmitted.

- *Metadata driven cache management* As object size varies in the cache, the cache policy needs to know not only the popularity of the content and its frequency of access, but also its size, to determine the best "bang for the buck" in keeping

the content. Having access to the content requests and the content size at the controller provide both.

5.3 Optimization problem formulation

To demonstrate the power of our approach, we focus now on network traffic optimization. The goal here is to optimize some metric of the network using content metadata that is gathered through content centric hooks in the network and is available to the controller. Let us split the problem into two parts. The first sub problem is storing the content in a cache: when the controller selects a cache, it will have to select a path to the cache, and will have to pick a cache that is not under stress already (for instance, from many write IOs form receiving other contents). Assuming the network will have a number of alternate paths between the ingress switch and the selected cache, this is an opportunity to use path diversity to optimize link utilization. We also assume that content metadata is available right after content enters the network (i.e. the ingress switch does DPI). Thus, here our objective is to minimize the maximum link utilization. This means that, we need to solve the following problem,

$$\begin{aligned} \min \max \quad & \sum_{p \in P} \sum_{e \in p} \frac{b_e + F}{c_e} \\ \text{subject to} \quad & b_e \leq c_e \end{aligned}$$

The second sub problem is content retrieval. The goal here is to minimize the delay a user will see when it requests content. This can be formulated as

$$\min_{e \in E} \quad \frac{F}{r_e}$$

The following table summarizes the notations used

b_e	Background traffic on link e
c_e	Capacity of link e
r_e	Rate of link e
F	Size of the content
P	Set of all paths between a source and a destination
E	Set of all links

Another interesting optimization problem that can be considered here is that of disk IO optimization. Given a number of caches in the network, each having a known amount of load at a given time, we might want to optimize on the disk writes over all of them and formulate the problem on that metric. Note that the actual optimization constraint to be used can vary on application requirements and is user programmable. Typically, these constraints will be programmed in the content management layer of the controller.

Given the architecture, we can show the end to end flow of content in the network, in a typical scenario. Consider a system as shown in figure 5.2. In this example we will assume that the objective is to optimize link bandwidth utilization by load balancing incoming content across redundant paths. We show both the schemes described in section **??** in the diagram. However, in a real implementation, it is sufficient to have one. Here are the steps the system will take, the initial few steps are very similar to that described in [13]. Note that this can be sub divided into three distinct steps: the first step is the setup phase where all devices connect to the controller and announce their capabilities, the second phase in the metadata gathering phase where network elements report back content metadata, this is used in the third phase for TE.

- All elements boot up and connect to the controller.
- Elements announce their capabilities to the controller. At this point, the controller has a map of the whole network and it also knows which nodes can extract metadata and cache content.
- Controller writes the special flow in all ingress switches, configuring them to extract content metadata. It also writes a "flow" to the cache asking it to report back content metadata.
- The client tries to setup a TCP connection to a server in the content provider network.
- An OpenFlow switch in the content network forwards the packets to the controller; which writes flows to redirect all packets from client to a proxy. At this stage, the client is transparently connected to the proxy.

- The client sends a GET request for a piece of content. Proxy parses the request and queries controller to see if that content is cached in the network.
- The first request for a content will be a cache miss since it is not already cached. Thus the controller returns nothing, the proxy forwards the request to the server in the provider network.
- The server sends back the content which reaches the ingress switch. The switch asks the controller where the content should be cached. This marks the explicit start of content.
- A special flow is pushed to each of the switches in the path and the content is cached. At this point, the controller knows where a content is cached.
- The ingress switch and the cached both returns content metadata. Now, the controller can map this metadata to a content name.
- If another consumer requests for the same content, the controller looks up it's cache dictionary by content name and the proxy redirects the request to the cache. Simultaneously, the controller uses the TE module to compute a path on which the content should be pushed to improve overall bandwidth utilization in the network. It writes flows to all switches to forward content.

5.4 Implementation discussions

As it is not an ICN architecture, the standard OpenFlow does not support all the functionality outlined in the earlier section. Here, we describe necessary modifications to the OpenFlow protocol to support the proposed mechanisms, since we view this as the fastest path to implementation. In this paper, we will focus only on content sent over HTTP since it forms the majority of Internet traffic. Other types of content can be addressed by extending the proposed mechanism in future work.

From a top level, network elements need to announce their capability of parsing (and caching) content metadata to the controller. Then the controller should be able to

tempDictionary = null
P = get all routes from ingress switch to selected cache
for $p\ in\ P$ **do**
    $tempCost = 0$
    for $e\ in\ p$ **do**
        $tempCost = tempCost + \frac{b_e+F}{c_e}$
    end for
    insert $\langle p, tempCost \rangle$ in tempDictionary
end for
return the path corresponding to the minimum cost in tempDictionary

Figure 5.1: Example of a path selection algorithm

write flows which will configure them to parse and send back metadata to the controller. Necessary modifications are described below:

5.4.1 Switch-controller handshake

During the handshake phase, the switch needs to announce its capability to parse content metadata. The controller can maintain a table of all switches that has advertised this capability. The switch announces its capabilities in a $OFPT\_FEATURES\_REPLY$ message So, we will need to add extra fields to it. the $ofp\_capabilities$ structure indicating capabilities to extract content metadata, cache content and proxy content.

5.4.2 Augmented flowmod

Once the controller connects to all the elements in the network, it will know which elements can extract metadata. The control plane will need to configure those elements by writing flowmods, asking them to parse content metadata. Thus, We will need an additional action on top of OpenFlow. We call it $EXTRACT\_METADATA$. A flowmod with this action will look like this

```
if ; actions=EXTRACT_METADATA,NORMAL
```

which essentially means, the switch will extract metadata from HTTP metadata, put it in a $PACKET\_IN$ message and send back to the controller. Later, the switch will perform a normal forwarding action on the packet.

We also introduce a new type of flowmod to OpenFlow. This format provides the ability to write flowmods which have an expiry condition:

```
if <conditions>; actions=<set of actions>
;until=<set of conditions>
```

Now, since the controller knows the length of a given content, it can use the per flow byte counter to set a condition for the *until* clause.

5.4.3 Switch to controller message

We are mostly interested in the content length which is encoded in HTTP headers (note that it is easy to extend this mechanism to extract other content metadata like mime type etc). Once a switch is configured to parse content, when it sees a HTTP packet, it will read the `Content-length` header and construct a tuple of the form $\langle contentname, contentsize, srcip, srcport, destip, destport \rangle$. This will be encapsulated in a $PACKET\_IN$ and sent back to the controller.

5.4.4 Content management layer implementation

Most OpenFlow controllers allow a dynamic module management system and a mechanism for modules to listen on $PACKET\_IN$ messages. Thus, the control management layer can be implemented as a module on a controller. It will subscribe to $PACKET\_IN$ messages. When it gets a packet, it will extract the information and discard the packet. This architecture allows the controller side to have multiple content management layers chained together.

The interaction between the switch and the controller is shown in the figure 5.2. Modified OpenFlow messages are marked with a star.

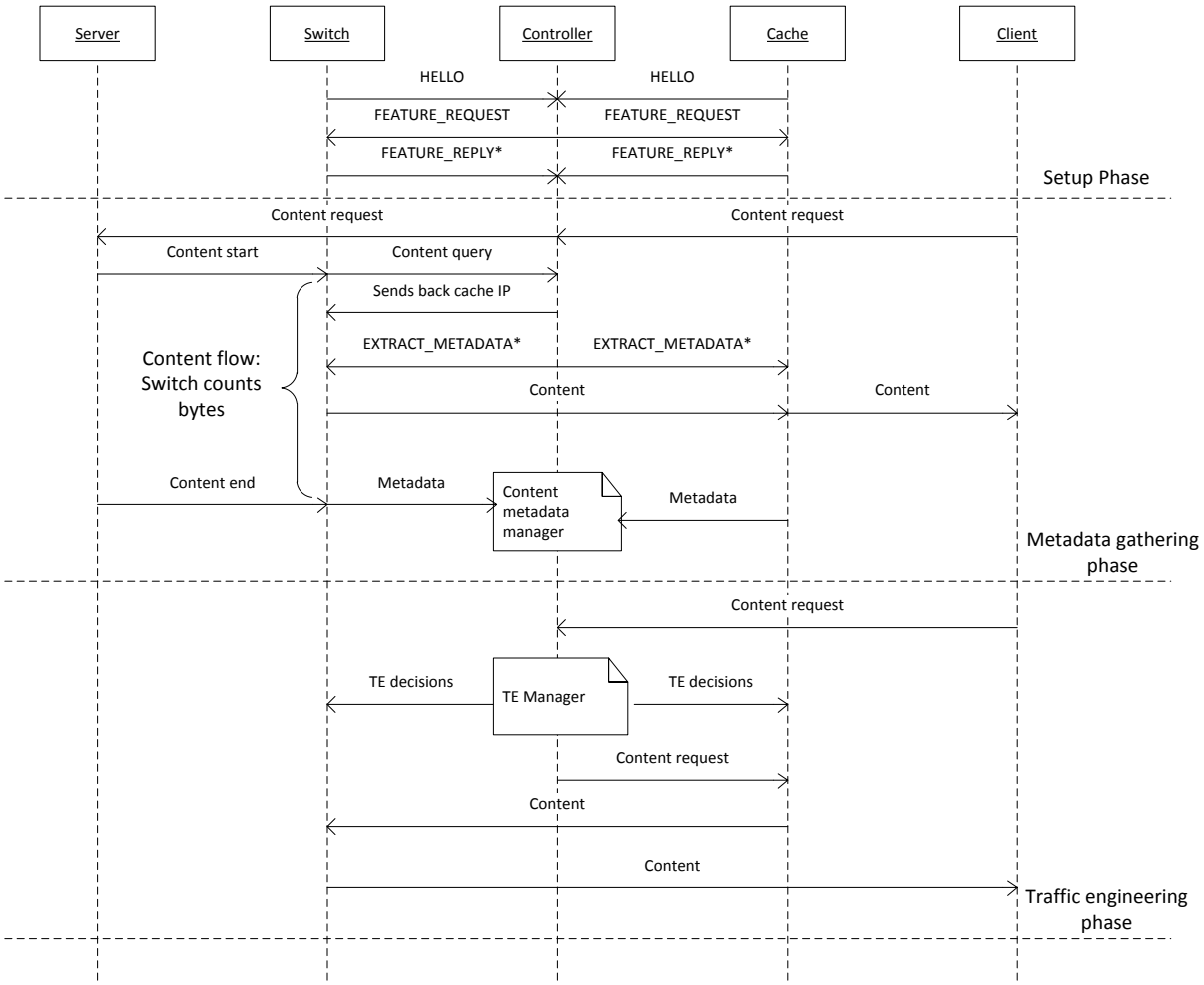

Figure 5.2: End to end flow in the system

Chapter 6

Implementation and Evaluation

To evaluate the merits of in-network storage virtualization, we implemented our proposed architecture on a small testbed. The testbed has a blade server (we would call it H) running Ubuntu. The server runs an instance of Open vSwitch which enables it to act as an OpenFlow switch. H runs three VMs each of which hosts the cache, the proxy and the client and also the FloodLight controller. This setup is placed in Santa Clara, CA. The major components of the system are described in the following sections:

6.1 Content management layer

We used FloodLight as the OpenFlow controller. FloodLight allows loading custom modules on the controller platform which can then write flows to all connected switches. We implemented the content management layer as a module to do content based forwarding on top of FloodLight. It subscribes to *PACKET_IN* events and maintains two data structures for lookup. The *requestDictionary* is used to map $\langle client, server \rangle$ pairs to request file names. This data structure can be queried using REST API to retrieve the file name corresponding to a request. The *cacheDictionary* holds mapping of content and it's location as the IP and port number of a cache.

6.2 Proxy

The proxy is written in pure Python and uses the *tproxy* library. The library provides methods to work with HTTP headers, note that there is no way to access any TCP or IP information in the proxy. The proxy uses the controller's REST API to communicate with it. It should be instantiated with the command

```
sudo tproxy <script.py> -b 0.0.0.0:<port number>
```

According to our design, the box running the proxy should run multiple instances of it on different ports. Each of those instances will proxy one $\langle client, server \rangle$ pair.

6.3 Cache

Our implementation of a cache is distinct from existing Internet caches in a number of ways: it can interface with an OpenFlow controller (running the content management module), on the flip side this does not implement usual caching protocols simply because it does not need to. Standard Internet caches see the request and if there is a miss, forwards it to the destination server. When the server sends back a response, it saves a copy and indexes it by the request metadata. Thus, they can setup a TCP connection with then server and use the socket interface to communicate. In our case, the cache sees only the response and not the request. Since it always get to hear only one side of the connection, it cannot have a TCP session with the server and so, cannot operate with a socket level abstraction. Thus, the cache must listen to a network interface and read packets from it. The cache has a number of distinct components as described below:

- **The Redis queue** There is a Redis server running in the backend which serves as a simple queuing mechanism. This is necessary to pass data (IP addresses) between the grabber module and the watchdog module. The grabber module (described below) can put IP addresses in the queue which can be read from the watchdog module.

- **The grabber module** The grabber module is responsible for listening to an interface and to read (and assemble) packets. It is written in C++ and uses the *libpcap* library. The executable takes an interface name as a command line argument and starts to listen on that interface. It collects packets with the same ACK numbers, when it sees a FIN packet, it extracts the ACK number and assembles all packets which has that ACK number. In this step, it discards duplicates. Note that, since there is no TCP connection, the cache won't know if

some packets are missing. It then extracts data from all those packets and writes back to a file in the disk with a default name. It also puts the sender's IP in the Redis queue.

- **The watchdog module** This module communicates with the controller using a set of REST calls. It is written in Python and uses the *inotify* library to listen on the cache directory for file write events. When the grabber module writes a file to the disk, the watchdog module is invoked. It calls the controller API to get the file name (using the IP from the Redis queue as parameter), it then strips all HTTP header from the file, changes it's name and writes it back. After the file is saved, it sends back an ACK to the controller indicating that the file is cached.

- **The cache server module** This module serves back content when a client requests. This is written in Python and is an extended version of *SimpleHTTPServer*.

6.4 Variation of access latency with file size

[16] reports that at the current moment 45% of all images served over the world wide web is of the JPEG format with an average size of $812Kb$. We placed 12 files of different sizes, from $2Kb$ to $6Mb$ on a web server located in New Brunswick, NJ. This range was empirically chosen so that the mean lies close to he half quartile. These files are then accessed from our client in Santa Clara, CA which opens up a regular browser and access the files over HTTP. We turned off browser cache for our experiments to ensure that the overall effect is only dues to our cache. FireBug [17] is used to measure content access delay in the two cases case, once when there is a cache miss (and the content gets stored in the cache) and the cache hit case (when the content is delivered from the cache).

We can also conduct a back-of-the-envelope delay analysis given Figure 3.2. In each case $TCP(A, B)$ represents the delay for establishing a TCP session between A and B and later to tear it down, $F(A, B)$ is the delay to download a file from A to B, $Delay(Proxy)$ is the processing delay at the proxy.

- Case 1 is when a client accesses a file directly without a cache or a proxy.

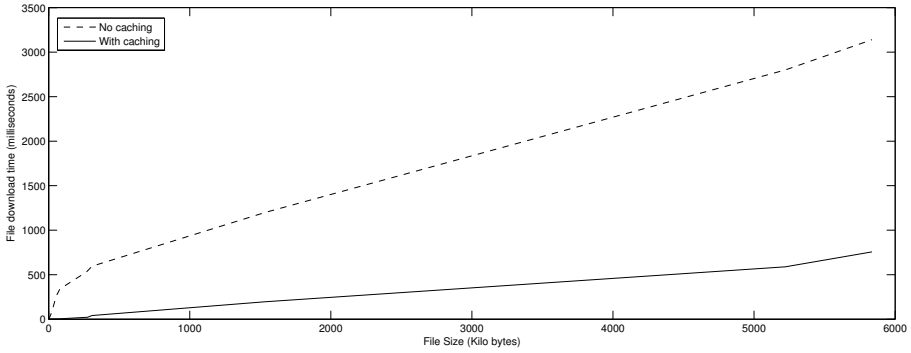

Figure 6.1: Variation of access latency with file size

In this case, the total delay is approximately equal to $TCP(Client, Server) + F(Server, Client)$.

- Case 2 is when the client uses a proxy (and an OpenFlow controller) to connect to the server. In this case, total delay is $TCP(Client, Proxy) + Delay(Proxy) + TCP(Proxy, Server) + F(Server, Proxy) + F(Proxy, Client)$

- Case 3 is our design with a proxy and a cache. Here delay $TCP(Client, Proxy) + Delay(Proxy) + TCP(Proxy, Cache) + F(Server, Proxy) + F(Proxy, Client)$

From the expressions, we see that case 1 has the highest delay followed by case 2 and case 3, assuming $Delay(Proxy)$ is negligible and that the proxy and cache is located in the same network as the client which makes file transfer delay between them very small. We noticed that, when content size is greater than $5kb$, all the expressions are dominated by $F(A, B)$ and $Delay(proxy)$ can be ignored. However, if the proxy is overloaded, this will increase resulting in case 2 and 3 having higher delay than case 1, which is not desirable. Those issues can be handled by offloading some of the control decisions at the ingress switch and/or proxy as discussed

6.5 Performance of traffic engineering

In this section, we will demonstrate that prior knowledge of content size can be used to decrease backlog in a link, which in turn results in lower network delay. The setup is the following: we have two parallel links between a source and a destination. Each link

has a capacity of 1kbps. Thus the total capacity of the system is 2kbps and at no point of time, the input should be more than this (otherwise the queue will go unstable). According to existing literature, we assume that content size is Pareto distributed [18]. Given a value of α and the mean load we will calculate the value of the shape parameter using the relation $b \leq mean\_load\frac{\alpha-1}{\alpha}$ so that the mean of the distribution is always less than 2. We assume deterministic arrival time of content, once every second from $t = 1$ to $t = 10000$. Traffic is allocated to each based on one of the following policies:

- *Policy 1* assumes that the content size is not known prior to allocating links. Thus, at any time instant, if both links are at full capacity, we will pick any one randomly. Otherwise, we pick the empty one.
- *Policy 2* assumes that we know the content size priori. At any time instant, we will pick the link with minimum backlog.

We study the variation of the gain in each case with increase in α from 1.1 to 2.5. For each value of α, we plot on Figure 6.2 the difference in % between the total backlog in the system under both policies. The size-aware policy 2 always reduces the amount of data waiting to be transmitted, and thus the delay in the system. For low loads, there is no need for optimization; for very high load, the links will always be highly backlogged and both policies are throughput optimal. We want to operate in a region where the link utilization is close to 1. Policy 2 shows signicant improvements in such a case.

The next experiment proves that policy 2 always results in lesser backlog than policy 1. This is shown by plotting the difference in queue backlog at each second which is always above zero. The average load in this case is $1.95kbps$ which is less than the maximum load of $2kbps$, thus the system is always stable. Note that this is not from a Pareto distribution.

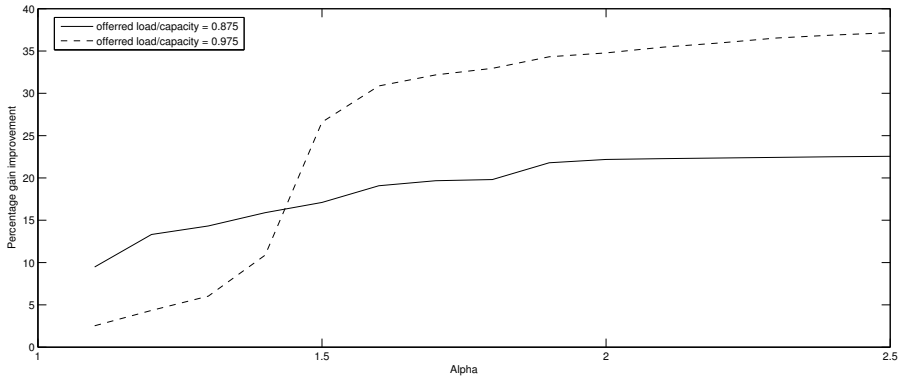

Figure 6.2: Variation of link backlog with increase in α: for the load/capacity of 0.95, the gain is up to 40% and 26% on average

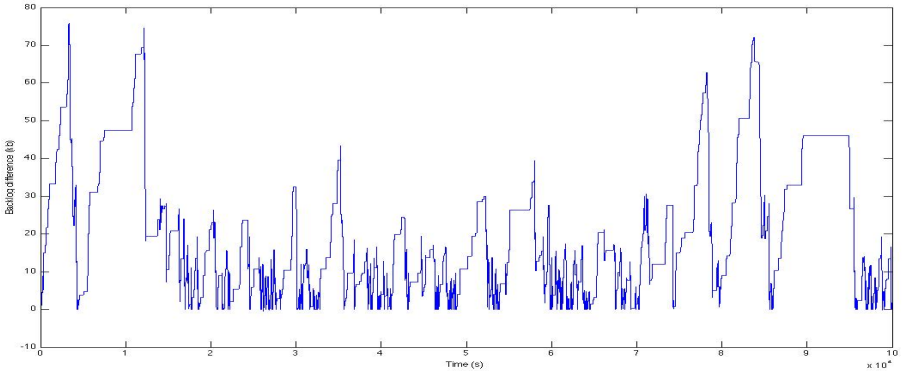

Figure 6.3: Policy 2 always results in lesser backlog than policy 1

Chapter 7

Conclusions and Future work

we have proposed and evaluated *ContentFlow*, a generalization of the SDN philosophy to work on the granularity of content rather than flow. This hybrid approach enables the end user to leverage the programmability and flexibility of SDNs coupled with the content management properties of ICNs. We have implemented *ContentFlow* in a small scale testbed, and a larger nation wide network which demonstrates that it is able to perform network-layer task at the content level seamlessly.

Some immediate questions point to directions for future work:

- How can we scale *ContentFlow* to handle large traffic loads?
- Can we improve the proxy hardware and can we distribute the content management functionality of the controller in order to reduce the latency?
- How does the caching policy impact the performance as observed by the end user?
- And can the set of primitives be expanded further beyond flows and contents, to include for instance some more complex workloads requiring synchronization of a wide set of network resources, including storage, compute and network?

Some open problems related to the traffic engineering aspects are to actually implement the proposed scheme in an actual switch and to deploy it in a larger scale. Another open question here is a choice of the optimization criteria for a given network. This can vary widely depending on a network operators constraints, a caching network operator might want to optimize disk writes while another operator might want to optimize link bandwidth usage in a data center. We would like to keep this externally configurable since the architecture is independent of the underlying optimization problem. We plan to handle all of this as a natural extension to this work.

References

[1] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, 1998.

[2] "Named data networking," http://named-data.org/, Aug. 2010.

[3] "ANR Connect, "content-oriented networking: a new experience for content transfer"," http://www.anr-connect.org/, Jan. 2011.

[4] S. Paul, R. Yates, D. Raychaudhuri, and J. Kurose, "The cache-and-forward network architecture for efficient mobile content delivery services in the future internet," in *First ITU-T Kaleidoscope Academic Conference*. IEEE, May 2008, pp. 367–374.

[5] "PURSUIT: Pursuing a pub/sub internet," http://www.fp7-pursuit.eu/, Sep. 2010.

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[7] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao, "Overview and principles of internet traffic engineering," RFC 3272, 2002.

[8] B. Fortz, J. Rexford, and M. Thorup, "Traffic engineering with traditional ip routing protocols," vol. 40, 2002, pp. 118–124.

[9] A. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection," in *INFOCOM*, 2011.

[10] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 19–19.

[11] B. Frank, I. Poese, G. Smaragdakis, S. Uhlig, and A. Feldmann, "Content-aware traffic engineering," in *arXiv:1202.1464 [cs.NI]*.

[12] G. S. Haiyong Xie and P. Wang, "Tecc: Towards collaborative in-network caching guided by traffic engineering," in *EEE INFOCOM (Mini Conference)*, 2011.

[13] A. Chanda and C. Westphal, "Content as a network primitive," *CoRR*, vol. abs/1212.3341, 2012.

[14] ——, "Contentflow: Mapping content to flows in software defined networks," *CoRR*, vol. abs/1302.1493, 2013.

[15] A. Chanda, C. Westphal, and D. Raychaudhuri, "Content based traffic engineering in software defined information centric networks," in *IEEE NOMEN*, 2013.

[16] "Http archive."

[17] "Firebug."

[18] M. F. Arlitt and C. L. Williamson, "Internet web servers: workload characterization and performance implications," in *IEEE/ACM Trans. Netw.*, 1997.