Traffic-Engineered Distribution of Multimedia Content

by

Khaled Diab

B.Sc., Cairo University, 2011

Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Science

© Khaled Diab 2019
SIMON FRASER UNIVERSITY
Summer 2019

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.
Approval

Name: Khaled Diab

Degree: Doctor of Philosophy

Title: Traffic-Engineered Distribution of Multimedia Content

Examinining Committee:

Chair: Steven Bergner
University Research Associate

Mohamed Hefeeda
Senior Supervisor
Professor

Jiangchuan Liu
Supervisor
Professor

Shervin Shirmohammadi
Internal Examiner
Adjunct Professor

Mostafa Ammar
External Examiner
Professor
School of Computer Science
Georgia Institute of Technology

Date Defended: 15 August, 2019
Abstract

The amount of video traffic transmitted over the Internet has been steadily increasing over the past decade. This is due to the recent interest in streaming high-definition (HD), 4K, and immersive videos to many users. Streaming such multimedia content at large scale faces multiple challenges. For example, streaming systems need to handle users heterogeneity and interactivities in varying network conditions. In addition, large-scale video streaming stresses the ISP network that carries the video traffic, because the ISP needs to carefully direct traffic flows through its network to satisfy various service level agreements. This increases the complexity of managing the network and system resources. To address these challenges, we first propose a novel client-based rate adaptation algorithm for streaming multiview videos. Our algorithm achieves high video quality, smooth playback and efficient bandwidth utilization. For example, it achieves up to 300% improvement in the average quality compared to the algorithm used by YouTube. Then, we propose an algorithm to efficiently solve the resource management problem in the emerging ISP-managed Content Distribution Networks (CDNs). Our solution achieves up to 64% reduction in the inter-domain traffic. To reduce the network load of live streaming sessions, ISPs use multicast to efficiently carry the traffic through their networks. Finally, we propose a label-based multicast forwarding approach to implement traffic-engineered multicast trees in ISP networks. We prove that the proposed approach is scalable as it imposes minimal state and processing overheads on routers. We implemented the proposed multicast approach in a high-speed network testbed, and our results show that it can support thousands of concurrent multicast sessions.

**Keywords:** Multi-view videos; adaptive video streaming; traffic engineering; telco-CDNs; multicast forwarding; efficient data planes.
To Dina, Jana, and my parents and brothers.
To my cousin Hazem.
Acknowledgements

I would like to express my sincere gratitude to my senior supervisor, Dr. Mohamed Hefeeda. When I met him in 2012, I only had a good knowledge in computer science and a mere interest in systems and networking. Ever since this moment, Dr. Hefeeda has been pushing my limits to gain the research skills to be a successful scientist. His continuous support, critical reviews and scientific knowledge as well as devotion and enthusiasm to research have contributed greatly to this thesis. I will be always indebted to Dr. Hefeeda.

I would like to express my gratitude to Dr. Jiangchuan Liu, my supervisor, for his valuable comments and discussions whenever I needed them. My thanks go to Dr. Shervin Shirmohammadi, my internal examiner, and Dr. Mostafa Ammar, my external examiner, for being on my committee and reviewing the thesis. I would like as well to thank Dr. Steven Bergner for taking the time to chair my thesis defense.

I would like to thank my colleagues at the Network Systems Lab who made this journey pleasant. Special thanks go to Dr. Joseph Peters, Ahmed Hamza, Kiana Calagari, Saleh Almowuena, Alaa Eldin Abdelaal, Omar Eltobgy, Omar Arafa, Mohammad Amin Arab, and Mohamed Hegazy. In addition, I am grateful for the help of all collaborators since I started my research. I am especially grateful to Mustafa Rafique for his valuable insights and discussions. I also like to thank the administrative and technical staff at the School of Computing Science for their help and support. My special thanks go to Lee Greenough, Johnny Zhang, Dana Brand, Stephen Nix, Chingtai Wong, Allison MacKinnon and Clare McKenna.

More importantly, I cannot find the right words to express my heartfelt gratitude to my parents and wife. I would like to thank my wonderful parents for their endless love and support throughout my life. I will be always grateful to my amazing wife, Dina, who has been supporting me to pursue my life and career goals. She has immersed me with her eternal love, support, and encouragement. This thesis would not have been possible without the support of my family.
Bibliographic Notes

Publications:


# Table of Contents

Approval ii  
Abstract iii  
Dedication iv  
Acknowledgements v  
Bibliographic Notes vi  
Table of Contents vii  
List of Tables x  
List of Figures xi  

1 Introduction  
   1.1 Overview .................................................. 1  
   1.2 Thesis Contributions .................................... 3  
      1.2.1 Adaptive Multiview Video Streaming over HTTP .... 3  
      1.2.2 Joint Content Distribution and Traffic Engineering in Telco-CDNs ... 4  
      1.2.3 Traffic-Engineered Multicast Forwarding .......... 5  
   1.3 Thesis Organization ....................................... 9  

2 Background  
   2.1 Introduction ............................................... 10  
   2.2 Adaptive Video Streaming .................................. 11  
      2.2.1 History of Video Streaming .......................... 11  
      2.2.2 HTTP Adaptive Streaming ............................ 11  
      2.2.3 Immersive Multimedia Content ...................... 12  
      2.2.4 Quality of Experience (QoE) in HAS Systems .......... 13  
   2.3 Rate Adaptation Algorithms ............................... 15  
      2.3.1 Single-view Videos ................................. 15  
      2.3.2 Immersive Videos .................................. 19
2.4 Content Distribution Networks (CDNs) ........................................... 21
  2.4.1 Challenges of CDNs .................................................................. 22
  2.4.2 Collaboration Space in CDNs ...................................................... 23
2.5 Multicast Systems ......................................................................... 26
  2.5.1 Application-layer Multicast ............................................................. 26
  2.5.2 Network-based Multicast ............................................................... 27

3 MASH: A Rate Adaptation Algorithm for Multiview Video Streaming over HTTP 30
  3.1 Introduction .................................................................................. 30
  3.2 Related Work ............................................................................... 32
  3.3 Overview ...................................................................................... 33
  3.4 View Switching Models ................................................................. 34
  3.5 Details of MASH .......................................................................... 36
  3.6 Analysis and Overheads of MASH .................................................... 38
  3.7 Evaluation .................................................................................... 39
    3.7.1 Experimental Setup .................................................................. 40
    3.7.2 MASH vs. YouTube Multiview Rate Adaptation ......................... 41
    3.7.3 Fairness and Comparisons vs. others ......................................... 44
  3.8 Summary ...................................................................................... 46

4 Joint Content Distribution and Traffic Engineering of Adaptive Videos in Telco-CDNs 48
  4.1 Introduction .................................................................................. 48
  4.2 Related Work ............................................................................... 50
  4.3 System Models and Problem Definition ................................................. 51
    4.3.1 Multimedia Content Model ....................................................... 51
    4.3.2 Telco-CDN Model .................................................................... 53
    4.3.3 Problem Definition ................................................................... 53
  4.4 Overview of CAD ........................................................................ 54
  4.5 Details of CAD ............................................................................ 55
  4.6 Evaluation .................................................................................... 59
    4.6.1 Implementation ....................................................................... 59
    4.6.2 Experimental Setup .................................................................. 60
    4.6.3 Results ................................................................................... 63
  4.7 Summary ...................................................................................... 68

5 Stateless Traffic-Engineered Multicast Forwarding 69
  5.1 Introduction .................................................................................. 69
  5.2 Related Work ............................................................................... 71
  5.3 System Model and Problem Definition ................................................. 73
  5.4 Proposed STEM System ................................................................ 74
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 4.1</td>
<td>Arrival rates of requests for different periods.</td>
<td>62</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Summary of current multicast forwarding systems.</td>
<td>72</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Label types in STEM. N is the number of routers, and I is the maximum number of interfaces per router.</td>
<td>76</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Packet latency (in $\mu$sec) measured in our testbed.</td>
<td>87</td>
</tr>
<tr>
<td>Table 6.1</td>
<td>Possible combinations of every link $(u \rightarrow v)$ based on its source $u$ and destination $v$ routers. The table shows the conditions to include links in the initial false positive candidate set. The examples are from Figure 6.3.</td>
<td>101</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>Validation of forwarding decisions made by our packet processing algorithm.</td>
<td>110</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Illustration of immersive videos: multiview and 360-degree.</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of a buffer-rate function used to determine segment bitrate based on current buffer occupancy $b(t)$.</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Illustration of different interconnection models between ISPs. Customers connect to the Internet through ISPs. Dotted lines correspond to the inter-domain traffic between ISPs.</td>
<td>21</td>
</tr>
<tr>
<td>2.4</td>
<td>The collaboration space among different actors in video streaming systems.</td>
<td>23</td>
</tr>
<tr>
<td>2.5</td>
<td>Illustration of the two approaches to implement multicast.</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>High-level overview of MASH. MASH runs at the client side, uses global and local view switching models, and constructs buffer-rate functions $f_i$'s to dynamically request segments from different views at different quality levels.</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>(a) Simple example of the stochastic switching model used by MASH. (b) The sigmoid function used in assigning importance factors to different views.</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>Proposed buffer-rate functions of views $V_i$ (active) and $V_j$ (inactive).</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>Setup of our testbed. Link characteristics (e.g., bandwidth and latency) are controlled using the Linux Traffic Control (tc) utility.</td>
<td>41</td>
</tr>
<tr>
<td>3.5</td>
<td>Comparison of MASH versus YouTube, for FQ (frequent view switching) scenario and bandwidth = 8 Mbps.</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>Average performance metrics of MASH versus YouTube across all experiments.</td>
<td>44</td>
</tr>
<tr>
<td>3.7</td>
<td>Load on the server to handle 100 concurrent users.</td>
<td>44</td>
</tr>
<tr>
<td>3.8</td>
<td>Comparison of MASH versus other rate adaptation algorithms.</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>A simplified view of the proposed CAD to manage telco-CDNs. CAD reduces the load on inter-ISP links and shortens the latency perceived by clients. Key new aspects include: jointly managing caching resources and network paths within the ISP, creating video representations on-demand, and enabling cooperation among caches.</td>
<td>52</td>
</tr>
<tr>
<td>4.2</td>
<td>The ISP topology used in our experiments. It is the AT&amp;T network in the East Coast of the US.</td>
<td>61</td>
</tr>
</tbody>
</table>
Figure 4.3 Probability distribution function (PDF) of the fraction of a video to be watched by clients. .......................................................... 62
Figure 4.4 Average inter-domain traffic, buffer occupancy at clients, and MLU when varying storage capacity. ........................................... 63
Figure 4.5 Time series of inter-domain traffic, buffer occupancy, and on-demand created traffic over the 3-day experiment period. ................. 65
Figure 4.6 Processing latency distributions of CAD to create representations on-demand under different storage capacities. ..................... 66
Figure 4.7 Average metrics of CAD and rCDN under different popularity patterns. Storage capacity is 10% of the library size. ...................... 67
Figure 5.1 High-level overview of STEM. ........................................... 74
Figure 5.2 Simple topology to illustrate different label types in STEM. Segment 1 → 7 does not follow the shortest path. ............................................ 76
Figure 5.3 Illustrative example of how labels in STEM represent the multicast tree in Figure 5.1. ................................................................. 84
Figure 5.4 Setup of our testbed. It has a NetFPGA that represents a core router in an ISP network and runs our algorithm. ................................. 85
Figure 5.5 Throughput of received traffic of concurrent multicast sessions from our testbed. We stress STEM by transmitting traffic that requires copying a range of labels for every packet. ............................................. 86
Figure 5.6 Average state size. #Routers=197 ........................................ 89
Figure 5.7 Analysis of state update rate. #Routers=197. ........................ 90
Figure 5.8 Analysis of label size and FSP savings of STEM. .................. 91
Figure 5.9 Analysis of processing overheads of STEM. ........................ 92
Figure 5.10 Running time of creating STEM labels at the controller. ........ 93
Figure 6.1 High-level overview of Helix. ........................................... 97
Figure 6.2 Overview of the Recursive Label Encoder (RLE). ................. 98
Figure 6.3 Illustrative example of the RLE algorithm. ........................... 108
Figure 6.4 Setup of our testbed. .......................................................... 109
Figure 6.5 Performance of Helix using the testbed. ............................... 111
Figure 6.6 The effect of label size on the session state for Helix and LIPSIN. ........ 114
Figure 6.7 Impact of receiver density on the session state for Helix and LIPSIN. The label size is 64 bytes. .................................................. 115
Figure 6.8 LIPSIN false positive overhead in three topologies. Helix does not produce any false positives. ........................................... 116
Figure 6.9 Performance of Helix vs OpenFlow. .................................... 117
Figure 6.10 Impact of filter size $B$ on the session state for multiple values of $K$. 118
Chapter 1

Introduction

1.1 Overview

Streaming videos over the Internet is a multi-billion-dollar industry and it is rapidly growing [1]. This is due to the proliferation of new types of multimedia content as well as the emergence of new video streaming applications. Both contribute to attracting millions of users to watch videos over the Internet. Specifically, there has been a significant interest in streaming full high-definition (HD) and 4K videos as well as immersive content such as multiview and 360-degree videos over the Internet. These new multimedia contents are becoming popular because they offer high-quality and unprecedented experience to users. For example, multiview videos allow users to explore scenes from different angles and perspectives, whereas 360-degree videos put users in virtual scenes that can be navigated.

In addition to the immersive videos, the online watching of live sessions over the Internet is becoming mainstream in many scenarios. The scale and dynamic nature of these applications are unprecedented. For instance, Facebook Live aims to stream millions of live sessions to millions of concurrent users [136, 114]. As another example, BBC reported a peak of 9.3 million TV users watching the Euro 2016 opening, and 2.3 million users watched that event through the BBC Internet streaming service [113]. Moreover, 2.2 million gamers share their live gameplay every month on Twitch [99].

As the popularity of immersive videos and live sessions increases, the amount of multimedia traffic distributed over the Internet increases at high rates. For example, the bit rate for a 4K video is more than double the HD video bit rate. The amount of live streaming traffic is expected to occupy 17% of the total Internet video traffic by 2022 [1]. Thus, major Internet service providers (ISPs) need to efficiently carry and manage larger volumes of multimedia traffic through their networks.

To generate revenues, content providers and ISPs need to maintain a high level of service quality for their customers. For content providers, streaming immersive videos to millions of users over the Internet is challenging due to the heterogeneous user interactivities and varying network conditions. In addition, the user expectation of better quality of experience is becoming higher. For example, users watching immersive content expect high-quality and smooth playback while interacting with
the video and navigating the scene. Some users leave streaming sessions if they experience long periods of interruptions or low quality [39]. In addition, major content providers require ISPs carrying their traffic to meet target quality metrics or service level agreements (SLAs). To achieve target SLAs for their customers, ISPs need to carefully manage and direct traffic flows of concurrent multimedia sessions through their networks. In many cases, the network paths are chosen (or engineered) to meet the requirements of an application/session/customer, and these chosen paths may not necessarily be the shortest paths computed by the routing protocol. For example, latency-sensitive sessions require passing through low-delay network paths, whereas popular sessions may need to go through paths with large available bandwidth. This is usually referred to as Traffic Engineering (TE), and it is more complex than just forwarding the traffic along the shortest paths.

Meanwhile, computer networks have evolved to include many capabilities such as programmable forwarding elements and processing resources at the edges. The control and data planes of networks have become programmable by high-level languages. In the control plane, service providers are able to dynamically update forwarding tables of routers to satisfy their TE objectives. This can be done through application-level APIs such as OpenFlow [98]. The programmability of control planes allows the ISP network to achieve high-level metrics for multimedia sessions (e.g., end-to-end latency). In the data plane, network designers are able to create new protocols and add them to existing programmable routers [14, 71]. This not only creates opportunities for optimizing multimedia traffic, but also for implementing new telemetry and security applications. Moreover, the network edge is becoming smarter by deploying processing and storage resources close to it [50]. This allows new (multimedia) applications to use these processing resources to satisfy their requirements.

The new demand of multimedia content as well as the advancements in network design have introduced new challenges and opportunities for large-scale distribution of multimedia content. In particular, the algorithms and systems that run the network infrastructure need to evolve to handle these demands, while considering the emergence of new principles such as network programmability and availability of processing resources at edges. One of the new challenges is to adaptively stream new multimedia content such as multiview videos over the Internet to thousands/millions of concurrent users. This is a complex task because of the varying network conditions as well as the level of user interactivity in multiview videos. Furthermore, it is hard to balance multiple, sometimes conflicting, performance metrics including quality level, quality variations, and smoothness of playback of different views.

Another challenge is to efficiently distribute multimedia content to caching sites deployed inside the ISP networks. Major ISPs, such as AT&T, deploy and manage content distribution networks (CDNs) inside their networks [11, 29]. Such CDNs are referred to as telco-CDNs. These emerging telco-CDNs offer new opportunities for optimizing the delivery of multimedia content, because not only can they jointly leverage the up-to-date information about the underlying ISP network and the characteristics of the multimedia objects, but they can also choose the appropriate paths for different sessions based on TE objectives and configure the network to enforce these decisions.
A third challenge is supporting millions of concurrent live multimedia streaming sessions. Delivering live multimedia sessions impose significant overhead on the network usage. Thus, the use of multicast is crucial to alleviate the load of increasing the number of live sessions. However, implementing multicast trees in the data plane is hard due to lack of scalability in current multicast forwarding systems. In addition, these systems cannot easily adapt to the dynamics of live sessions in terms of routers joining/leaving, or the dynamics of the network such as link failures.

To address these challenges, we propose new algorithms and systems to: (1) adaptively stream multiview videos over HTTP [34], (2) distribute multimedia content to caching sites inside telco-CDNs while satisfying predefined TE objectives [36], and (3) deliver traffic-engineered flows of live multimedia sessions through multicast inside ISP networks [37, 35].

1.2 Thesis Contributions

We summarize the contributions of this thesis in the following.

1.2.1 Adaptive Multiview Video Streaming over HTTP

The first problem we consider is to adaptively stream multiview videos using HTTP Adaptive Streaming standards such as DASH [123]. In DASH, a rate adaptation algorithm is used by each client to dynamically request segments with the most suitable quality for the current conditions. The rate adaptation algorithm needs to balance multiple performance metrics including quality level, quality variations, and smoothness of playback [39, 91].

Multiview videos consist of several views of the same scene but captured by multiple cameras at different angles. These videos offer unprecedented experience by allowing users to explore scenes from different angles and perspectives. For example, YouTube [3] released a small-scale experiment to stream multiview content, in which a user can experience the scene (a concert in that experiment) from different perspectives by switching among various views. Two of the main challenges in streaming this complex multiview video content over the Internet are adaptation to dynamic network conditions and supporting user interactivity.

Unlike single-view videos, multiview videos require much more complex rate adaptation algorithms. This is because these algorithms need to fetch segments for active views as well as other views to enable users to smoothly navigate across views. In addition, naively fetching data from all possible views can lead to significant waste in network bandwidth. On the other hand, fetching data from only the active view and nothing from other views will introduce long delays when a user switches to another view, which can damage the immersive experience and lead to user dissatisfaction. Furthermore, users interact with multiview videos in different ways, based on their own interests and perspectives. Thus, predicting the view that may be needed next is not straightforward. These complications are all added to the challenge of handling the network dynamics.

To address these challenges, we propose a novel Multiview Adaptive Streaming over HTTP (MASH) algorithm. MASH introduces a new perspective to the rate adaptation problem in multiview
video streaming systems: it constructs probabilistic view switching models and utilizes these models in dynamically selecting segments of various views at different qualities, such that the quality of the videos are maximized while the network bandwidth is not wasted.

Specifically, we make the following contributions to address the rate adaptation problem of multiview videos [34]. We present view switching models based on Markov chains to capture user activities. For each multiview video, MASH constructs global and local view switching models. The global model captures user activities across all streaming sessions seen by the server so far, while the local model understands user activity during the current streaming session only. MASH combines the two models to dynamically weigh the importance of each view given the one being watched.

We propose a new buffer-based rate adaptation algorithm for multiview video streaming systems. MASH uses the new buffer-based approach to select segments of different views according to their relative importance. We show that the proposed algorithm and view switching models impose negligible overheads and we analyze their convergence properties.

We developed a multiview video player and implemented our rate adaptation algorithm and view switching models in it. We conducted extensive empirical study to compare MASH versus the algorithm used by YouTube for multiview videos. The results show substantial improvements across multiple performance metrics and in different network conditions. For example, MASH achieves up to 300% improvement in the average rendering quality and up to 200% improvement in the prefetching efficiency compared to the YouTube algorithm. In addition, our results show that MASH achieves fairness across concurrent sessions, and it does not overload the streaming server. Furthermore, we compare MASH versus other rate adaptation algorithms, which are derived from current rate adaptation algorithms for single-view videos, and we show that MASH outperforms all of them.

1.2.2 Joint Content Distribution and Traffic Engineering in Telco-CDNs

Current CDNs often infer network conditions inside ISPs through complex measurement methods [107, 50] to improve user-perceived quality. However, CDNs cannot control the traffic flows and what paths they take inside ISP networks, which ultimately carry the actual traffic. CDNs may not even precisely know the underlying network topology and the current traffic situation on links and switches in the ISP network. Thus, decisions made by CDNs may negatively impact the load on various links in the ISP networks [94], especially links between different network domains (called inter-ISP links), which are costly [105]. This may trigger ISPs to adjust traffic routing inside their networks, which in turn, could impact the performance of CDNs.

Multiple ISP-CDN collaboration models have been proposed to reduce the mismatch between the goals of CDNs and ISPs. These models vary from defining interfaces that ISP and CDN can use to share information [49], to allowing content providers to deploy caches within ISPs (e.g., Netflix OpenConnect [103]). While useful, these approaches can only provide partial solutions. For example, OpenConnect caches [103] can provide local access to popular content within the ISP, but they cannot control the network paths taken to carry the multimedia sessions. The difficulties
of enabling ISP-CDN collaboration and the potential business opportunities motivated major ISPs, e.g., AT&T, to deploy and manage CDNs inside their networks [11, 29] (referred to as telco-CDNs).

The second problem we address in the thesis is to serve user requests from telco-CDNs by allocating the storage and processing resources at each caching site to the different video representations as well as directing the traffic flows of the video sessions through the ISP network such that the cost incurred by the telco-CDN is minimized (in terms of the amount of inter-ISP traffic) and the user-perceived quality is optimized (in terms of end-to-end latency). Managing telco-CDNs is, however, a complex task, because it requires concurrently managing resources at the network layer (TE) and system layer (processing and storage capacities of caches), while supporting the adaptive nature and skewed popularity of multimedia content.

We make the following contributions to address the resource management problem in telco-CDNs [36]. We propose an algorithm to efficiently solve the resource management problem in the emerging telco-CDNs, which we refer to as CAD (short for Cooperative Active Distribution). CAD has two goals: (i) reducing the cost incurred by ISPs, and (ii) improving the quality of multimedia sessions by reducing end-to-end latency. CAD achieves these goals by serving as much as possible of the multimedia traffic locally within the ISP, while carefully engineering the traffic paths to avoid creating bottlenecks in the network. CAD can significantly improve the performance of multimedia streaming systems in scenarios where ISPs and content providers cooperate (e.g., Netflix OpenConnect [103]), and when ISPs own their streaming services such as AT&T U-Verse and Bell Canada. Both scenarios are increasingly seeing wider adoption and deployment in practice.

We have implemented CAD and evaluated it on top of the Mininet [61] network emulation environment, which processes and carries real traffic. We implemented caching servers that stream multimedia content using the widely-used DASH protocol [123]. We used virtual switches to realize the TE aspects of CAD, which are enforced using SDN (Software-defined Networking) rules. We used an actual ISP topology and conducted extensive experiments to assess the performance of CAD and compare it against the closest work in the literature [119]. Our results show substantial improvements across multiple performance metrics. For example, CAD achieves up to 64% reduction in the inter-domain traffic and 14% improvement in the buffer occupancy (i.e., number of buffered segments) at clients compared to the algorithm in [119]. The buffer occupancy indicates a better quality of experience (QoE) for users in adaptive streaming [73]. Our results also show that CAD does not overload the intra-domain links, which ensures the stability of the inter-domain routing policies employed by the ISP [127].

1.2.3 Traffic-Engineered Multicast Forwarding

Large-scale live streaming applications, such as Facebook Live, have introduced a renewed interest in scalable multicast services. The amount of live streaming traffic has been increasing at high rates, and it is expected to occupy 17% of the total Internet video traffic by 2022 [1]. To reduce the network load of such applications, ISPs use multicast to efficiently carry the traffic through their networks. Moreover, traffic flows of these multicast sessions are engineered through the ISP network to satisfy
various TE objectives such as minimizing the maximum link utilization [69, 57]. Thus, the traffic flow of a multicast session may not always be forwarded on the shortest network paths within the ISP. Instead, the traffic flow of a multicast session is forwarded on what we call a traffic-engineered multicast tree, which is a distribution tree created over routers in the ISP to reach all end users of the multicast session, while achieving the TE objectives. As an evidence of the renewed interest in multicast, Comcast has deployed Xfinity and BT has deployed YouView multicast services.

The next problem we consider in the thesis is to realize traffic-engineered multicast trees in the network and support their dynamic changes over time. To implement such trees in the data plane, ISPs need to deploy a multicast forwarding system. However, designing efficient multicast forwarding systems is quite challenging, as it needs to address three main issues: (i) scalability, (ii) efficiency, and (iii) generality. The scalability has been an issue since the introduction of IP multicast a few decades ago. The main concern is the state that needs to be maintained at routers and the frequency and cost of updating this state. For example, current protocols that support TE, such as OpenFlow, require keeping state (match-action rules) at every router in the tree of the multicast session. As a result, the required state at routers grows linearly with the number of multicast sessions.

Efficiency of a multicast forwarding system means that routers should forward packets on and only on links of the multicast tree and should not result in any forwarding loops in the network. Although an obvious requirement, efficiency is not easily achieved nor always guaranteed by current multicast systems in the literature. For example, the LIPSIN system [81] may forward packets on links that do not belong to the multicast tree (referred to as false positives) and can also introduce loops in the network. Finally, the generality of a multicast system indicates that it can support different tree sizes, traffic patterns, and network topologies. Again, not all multicast systems in the literature possess this property. For example, the Elmo multicast system [118] is designed for a specific topology for data center networks and thus cannot be used with other network topologies.

We propose a label-based forwarding approach to address these challenges. The key idea is to represent the multicast tree of a session as a label attached to packets of that session. As a result, state at routers can be reduced or totally eliminated. The proposed approach consists of two components: centralized controller and packet processing algorithm deployed at routers. The controller calculates a label for each multicast tree, and instructs an ingress router to attach the computed label to packets of that session. The data plane algorithm processes labels to forward or duplicate packets on links belonging to the tree.

Various ISP networks are built with different design choices due to non-technical reasons such as deployment history and budget constraints, or technical issues such as performance and security. To address the heterogeneity of ISP network deployments, we propose two multicast forwarding systems to target different network environments. The first environment is when the controller cannot directly instruct core routers to store additional state or forwarding rules. This can happen because core routers do not have sufficient memory resources, these routers do not implement protocols to update their state, or the network administrator does not deploy such protocols for security rea-
sons [21]. In this scenario, we design a multicast forwarding system (called STEM) that creates a stack of labels for every multicast session, and instructs an ingress router at the network edge to attach these labels to packets of the multicast session.

The second network environment we target is when the controller can instruct core routers to update their state. For such environment, we propose a multicast forwarding system called Helix. The proposed system splits forwarding information about every multicast tree between labels attached to packets, and state maintained at some routers. We design Helix to produce a compact and fixed-size label for each multicast session so that it can be efficiently processed at routers.

**STEM: Stateless Traffic-Engineered Multicast Forwarding**

The first network environment we consider is when the controller cannot update the state at core routers. We design STEM to entirely move the forwarding state of a multicast tree from routers to labels on packets. As a result, STEM is scalable because it does not require state at routers and does not impose significant communication overheads. It only updates one (ingress) router when a tree changes. This allows fast adaptation to network dynamics such as routers joining/leaving sessions and link failures.

We make the following contributions [37] to address the challenges of stateless multicast forwarding systems. We propose four label types to represent a given traffic-engineered tree. The first two types are Forward Shortest Path (FSP) and Forward Traffic Engineered (FTE) labels. They are used by routers to forward packets on paths that have no branches. The other two types are Multicast (MCT) and Copy (CPY) labels. MCT label instructs routers to duplicate a packet on multiple interfaces, and CPY label copies a subset of the labels. STEM strikes a balance between label overhead on packets and parsing overhead at routers. In particular, every label type has a fixed size to minimize the total label overhead while achieving fast label processing.

We develop a control-plane algorithm to calculate a set of labels for each multicast tree. The algorithm divides the tree into segments and branching points. The goal is to calculate a set of ordered labels. The algorithm calculates FSP and FTE labels for every segment, and MCT and CPY labels at branching points. The label order is important because it reflects which routers process what labels. We propose a packet processing algorithm, deployed at routers, to process the attached labels to a packet. The algorithm forwards/duplicates packets on correct interfaces by removing labels that are not needed by next routers, and copying a subset of labels at branching points.

We implemented STEM in a hardware testbed using NetFPGA. Our results show that STEM can support high-speed links carrying thousands of concurrent multicast sessions, and it requires less than 3% of the available logic resources per port on the NetFPGA. In addition, we compared STEM against a rule-based approach implemented using OpenFlow and a label-based approach [81] in simulations using real ISP topologies with different sizes. Our results show that STEM reduces the number of routers to be updated when a multicast session changes by up to 30X and 10X compared to the rule-based and the label-based approaches, respectively.
Helix: Efficient Multicast Forwarding System

We next consider the ISP network environment where protocols that enable the controller to update the state at core routers are deployed in the network. We design Helix to utilize this capability while addressing the challenges of multicast forwarding systems. The key idea of Helix is to systematically split the information about a multicast tree into two parts: constant-size label attached to packets and small state at some routers in the tree. This information-split architecture leads to substantial improvements in the data plane performance, as well as allows Helix to strike a balance between the communication (label) overhead and the memory and processing overheads imposed on routers to handle labels and forward traffic on the multicast tree.

The controller of Helix uses a probabilistic set membership data structure (denoted by filter) to encode the tree link IDs in a relatively small label. This encoding is done using hash functions as in the case of Bloom and Cuckoo filters [17, 43]. Since these filters trade off membership accuracy for space efficiency, they may result in false positives, which occur when some links that do not belong to the multicast tree are incorrectly included in the computed filter. False positives waste network resources, overload routers, and could create loops. In Helix, these false positives are maintained as state at corresponding routers. In our design, labels do not change as packets traverse the network. Furthermore, labels are carefully structured so that they can be processed at high speed without overloading routers.

We make the following contributions [35] to address the challenges of multicast forwarding. We design a control-plane algorithm, called the Recursive Label Encoder (RLE), to produce a label and state. RLE recursively encodes link IDs of a given tree in $K$ rounds. Each round uses a filter to encode the input link IDs in a small label. The inputs to each round are carefully selected to successively reduce the number of entries in the final state, ensure efficiency of forwarding, and facilitate the packet processing at routers. The label of the session is created by concatenating all intermediate labels produced from the $K$ rounds. The state remained after the $K$th call is the final state used in the forwarding decisions. We develop an efficient packet processing algorithm that runs in a small and fixed number of clock cycles at routers (regardless of the label size). The algorithm leverages the structure of RLE to forward packets on and only on interfaces that belong to the multicast tree.

We have implemented Helix in a NetFPGA testbed to show its feasibility and practicality. Our experiments show that Helix can easily provide line-rate throughput, and it consumes a negligible amount of the NetFPGA resources (less than 1% of the hardware resources). We have also compared Helix in simulations against the closest multicast forwarding systems in the literature using real ISP topologies. Our simulation results show that Helix is efficient and scalable, and it substantially outperforms the closest systems across all considered performance metrics.
1.3 Thesis Organization

The chapters of this thesis are organized as follows. We first present necessary background on adaptive video streaming algorithms, content distribution networks, and multicast systems in Chapter 2. In Chapter 3, we present MASH, the client-based adaptation algorithm for multiview videos. In Chapter 4, we address the problem of joint traffic engineering and content distribution of adaptive videos in telco-CDNs. In Chapter 5, we present a label-based multicast forwarding system called STEM to implement traffic-engineered trees in ISP networks where the controller cannot instruct core routers to update their state. In Chapter 6, we address network environments that enable the controller to update the state at core routers. We describe the design of a novel multicast forwarding system, called Helix, that splits information about trees into a label and state maintained at some routers. We conclude the thesis and discuss future research directions in Chapter 7.
Chapter 2

Background

In this chapter we discuss the history of video streaming protocols, the basics of HTTP adaptive streaming (HAS), and the recent rate adaptation algorithms that choose video quality. We next discuss the need for content distribution networks (CDNs), and describe major challenges that face traditional CDNs. Then, we discuss the distribution systems from the angle of the collaboration space between CDNs, ISPs, content providers and users. Finally, we describe different approaches to implement multicast forwarding systems in ISP networks.

2.1 Introduction

Immersive multimedia content offer unprecedented experience by allowing users to explore scenes from different angles and perspectives. Achieving high quality of experience (QoE) to stream immersive videos over HTTP is challenging. On one hand, the Internet is a best-effort transport network. That is, there are no guarantees on the end-to-end performance, and the network conditions vary in a short period of time. In addition, HAS poses many challenges to achieve high QoE in such dynamic and diverse network. This is because of the discrete nature of HAS content, the lack of full observability of HAS clients, and the effects of wrong decisions on the future QoE. On the other hand, the user expectation of short-delay interactivity and the big data volume in immersive videos pose more challenges. In particular, there is a mismatch between smoothly exploring the immersive content and the bandwidth requirements. That is, fetching all views, that the user does not watch, results in wasting bandwidth. Moreover, fetching one view, that the user currently watches, results in playback freezes when the user explores different views. To address these challenges, robust quality selection algorithms are required to choose the best quality level that matches the varying network condition and the diverse user interactions.

The next challenge in video streaming systems is to bring the content closer to end users. CDNs make agreements with major ISPs to connect their servers to the ISP networks, and they store popular videos on these servers. Although CDNs help reduce the service delay and interdomain traffic, they are not aware of the varying conditions of ISP networks (e.g., available bandwidth and link failures). Thus, they apply expensive techniques to estimate the state of the underlying network
that carries the video traffic. Yet CDNs cannot control the network paths taken by video sessions, which can reduce the quality of experience and increase the network overheads. To address this challenge, many collaboration techniques were proposed to improve the performance of content distribution systems.

2.2 Adaptive Video Streaming

2.2.1 History of Video Streaming

Video streaming has been one of the main applications of the Internet. Since the start of the Internet, video streaming has gone through three main phases [88]: (1) client-server architecture based on the User Datagram Protocol (UDP), (2) Peer-to-peer (P2P) streaming, and (3) Hypertext Transfer Protocol (HTTP) adaptive streaming (HAS).

During 1990’s to early 2000’s, it was believed that UDP was a good fit for streaming, since UDP could utilize the network throughput without the need for the slow start and congestion control of the Transmission Control Protocol (TCP). In UDP streaming, the server hosts video content, and the client requests and renders video frames. Both the academia and the industry proposed and deployed protocols such as: (1) Real-time Transport Protocol (RTP) on top of UDP for video data framing and handling jitter and packet loss, (2) RTP Control Protocol (RTCP) for session monitoring and quality statistics, and (3) Real Time Streaming Protocol (RTSP) to support session control such as play, stop and pause. RTP/RTCP/RTSP protocols did not provide the required scalability when the number of users increases for two reasons. First, the server was actively involved in every streaming session. Second, the limited network and server resources at that time.

During the 2000’s, Peer-to-peer (P2P) streaming systems [62, 145] were proposed to mitigate the scalability issues of client-server architectures. In this paradigm, the end users acted as peers where every peer is both a server (that hosts content), and a client (that requests content). As a result, content discovery protocols between peers were needed so that peers know where every content is hosted. P2P systems, however, suffered from three drawbacks. First, the video quality was not necessarily guaranteed due to peers failure and leaving the networks. Second, P2P could result in traffic imbalance [142] at the network level, where the load could concentrate in one region. Third, P2P users needed special applications [88] which hindered its adoption and client heterogeneity.

The third phase is using HTTP for video streaming (referred to as HAS). Compared to its predecessors, HAS provides many advantages such as scalability, ease of deployment and fault-tolerance. However, HAS poses multiple challenges to stream both single-view and next-generation videos. In the following, we describe HAS, next-generation content, challenges of HAS, and the design space of its algorithms.

2.2.2 HTTP Adaptive Streaming

To solve the scalability and deployment issues of RTP and P2P, both the academia and industry focused their efforts towards HTTP-based streaming. The academia has proposed HTTP-based
streaming protocols such as the Dynamic and Adaptive Streaming over HTTP (DASH) [123]. In addition, current popular streaming services such as Netflix and YouTube use DASH as its main transport protocol. In this report, we refer to DASH-like protocols as HAS.

The video in HAS is encoded into multiple quality levels at different bitrates. Each quality level is divided into segments of equal length, and stored at an HTTP server. During the streaming session, clients request segments from the HTTP server. Requested segments are decoded and pushed to the video frame buffer at the client. The frame buffer has a buffer occupancy in time units (i.e., seconds). The buffer compensates for the network variations during the streaming session.

HAS has many advantages compared to the aforementioned protocols. First, HTTP provides a uniform way to access content (i.e., both text and multimedia), and enables such traffic to seamlessly pass through Network Address Translators (NATs) and firewalls [111] which are already deployed in many Internet Service Providers (ISPs). Second, the server-side has become a simple stateless server or cache. As a result, current Content Distribution Networks (CDNs) do not require major changes in their infrastructures, which facilitates the deployment and growth of video streaming providers. Third, using CDNs reduces the network load and latency for ISPs [85]. Finally, a client can fetch video segments from multiple servers or caches, which allows load balancing for content providers and CDNs [93, 92], and supports replication to mitigate failures.

Compared to traditional HTTP workloads that require either low latency or high average bandwidth, video streaming over HTTP requires continuous high Quality of Experience (QoE) over a long period of time (e.g., in order of minutes) [93]. While HAS promises ease of deployment and scalability, it comes with the challenge of how to choose the quality level that achieves high QoE in the face of the Internet network dynamics.

### 2.2.3 Immersive Multimedia Content

As discussed earlier, HAS temporally divides the single-view video to multiple segments. Next-generation videos add a spatial requirement as well. That is, at any time instance, the user is able to watch different angles of the same scene. We categorize immersive videos to multiview and 360-degree videos as follows.

**Multiview Videos.** In these systems, multiple cameras capture different angles of the same scene as depicted in Figure 2.1a. During streaming sessions, users are able to switch between multiple views to navigate the scene from various angles.

Different types of multiview videos can be generated based on the camera capabilities. We consider three types: 2D multiview, 3D multiview, and free-viewpoint video (FVV). In 2D multiview videos, each camera captures a single texture view. The views are then encoded either independently or based on the spatial dependencies between the views.

3D multiview videos are captured using video plus depth cameras. When the view is rendered at the client side, the depth is used to generate a 3D scene. In addition, these cameras enable generating a new type of content called free-viewpoint video (FVV). In FVV, virtual views are created when the user navigates to a view that was not captured by a real camera. View synthesis can be done using
Multiple operations are required to produce the final multiview video such as camera alignment, calibration and synchronization, image rectification, and encoding techniques.

**360-degree Videos.** Recently, there has been a huge interest from end-users to watch 360-degree videos (also called omnidirectional videos) over the Internet for two reasons. First, the hardware industry has advanced and enabled many vendors to manufacture devices to render 360-degree videos. Second, the availability of capturing devices and content generation algorithms [20] resulted in increase in 360-degree content. To watch 360-degree videos, the end-user often wears a head-mounted device (HMD) that renders a virtual scene. End-users explore this virtual scene by performing head rotations. The current view being watched (called a viewport) is defined by the rotation angles and the field of view (FoV) of the device.

To prepare a 360-degree video, the views captured from multiple cameras (shown in Figure 2.1b) are stitched and projected from a sphere to a plane [41]. Examples of these projections are equirectangular, cubemap and pyramid. Every projection has its own advantages and drawbacks. For instance, equirectangular projections are widely used and can be viewed without special player. However, they result in redundant pixels and may distort the original view.

Compared to 2D videos, streaming multiview and 360-degree videos requires more bandwidth due to large amount of data, and high resolution and frame rate [41]. In addition, the users may quickly move their heads, which result in fetching additional video data without rendering it. Compared to multiview videos, 360-degree videos provide panoramic experience, where users watch videos as if they were at the center of the scene.

### 2.2.4 Quality of Experience (QoE) in HAS Systems

User studies [83, 39, 101, 134] analyzed performance metrics that should be optimized for single-view and immersive videos to achieve high QoE, hence, gaining better user engagement. User en-
gagement not only affects user satisfaction, but also the ability of content providers to generate more revenues. We describe the most common metrics in the following:

- **Rendering Quality** which is measured by the average bitrate of the watched video. Users who watch higher quality videos are likely to watch the video longer [39].

- **Rate of Buffering Events** is the number of buffering events divided by the session duration. Dobrian et al. [39] showed that a 1% increase in the rate of rebuffering reduces the total play time by more than 3 minutes.

- **Rate of Temporal Quality Switches** is the rate at which the user perceives different quality levels over time. Mok et al. [101] show that switching between different quality levels over short period of time reduces overall user experience.

- **Startup Delay** is the time required to fetch and render the first frame of the video. Krishnan and Sitaraman [83] show that users start leaving a streaming session if it takes more than 2 seconds to start. Every additional delay of 1 second results in a 5.8% increase in the rate of leaving the session.

- **Spatial Quality Variance** is defined in tiled video streaming (e.g., Virtual Reality (VR)) as the difference between the quality levels of different tiles in the frame. Wang et al. [134] shows that QoE degrades if the spatial quality of tiled videos is not smooth.

Achieving high QoE for video streaming over HTTP is, however, challenging compared to other Web transfers. First, the discrete nature of HAS quality levels will result in on-off patterns of the link utilization. As a result, the client does not have a truthful measurement of the bandwidth during off periods [78, 91, 72]. Second, most of these QoE metrics are conflicting by nature. For instance, increasing quality level may result in rebuffering events. Sensitivity to network throughput may result in too many quality switches. Third, inaccurate network estimation results variable and low quality video streaming [72, 91]. Fourth, there is hidden state to the client at lower levels of the network such as the number of competing flows [126]. Finally, interactivity in next-generation videos (i.e., multiview and VR) adds a new layer of complexity. This is because the client needs to fetch segments for multiple views/videos to enable the user to smoothly navigate across views. In addition, the video may have many possible views, and naively fetching segments from all of them can lead to significant waste in network bandwidth. On the other hand, fetching segments from only one view and nothing from other views will introduce long delays when a user switches to another view, which can damage the immersive experience and lead to user dissatisfaction. Furthermore, users interact with these videos in different ways, based on their own interests and perspectives. This results in two problems. First, predicting the view that may be needed next is not straightforward [34]. Second, the corresponding frame buffers are usually small to compensate for network dynamics and user interactivity [143].
2.3 Rate Adaptation Algorithms

To handle varying network conditions and diverse user interactions, a quality selection algorithm is required to choose, for every client, the quality level of every video segment (i.e., the temporal aspect in HAS). For next-generation videos, this algorithm needs to consider per-view or per-tile quality level as well (i.e., the spatial aspect in HAS). In this report, we survey different quality selection algorithms based on the following three dimensions.

- **Content type**: We divide video types to single-view and next-generation videos. As mentioned above, next-generation videos share the same challenges of single-view videos, and add user interactivity as a new dimension of complexity.

- **Selection mechanism**: Different algorithms selects the quality levels according to different input signals. We divide the selection mechanism to three types: rate-based (i.e., only estimates the network bandwidth), buffer-based (i.e., only observes the buffer occupancy), or hybrid (i.e., look at both network estimation and buffer occupancy).

- **Deployment option**: We consider two options to deploy the quality selection algorithms. The first option is to deploy a distributed algorithm that runs independently at every client during streaming sessions. We refer to these algorithms as rate adaptation algorithms. The second option is a centralized controller that runs in a server, and communicates the quality levels to the clients. We call these controllers video control planes.

We note that these dimensions are independent. For example, a content provider may choose to design buffer-based adaptation algorithm that runs independently at every client to stream 2D videos. While other providers may deploy a centralized control plane using hybrid information from every client to streaming VR videos. However, we do not claim that these are complete dimensions. For example, we do not discuss adaptive algorithms that are designed specifically for mobile streaming [137].

In the following sections, we survey client-based adaptation algorithms for single-view and next-generation video streaming.

2.3.1 Single-view Videos

We categorize rate adaptation algorithms to rate-based [78, 91, 101, 131], buffer-based [73] and hybrid [148] approaches. In rate-based approaches, the algorithm estimates the network capacity and chooses suitable segments accordingly. Accurate network capacity estimation is, however, difficult [72]. Buffer-based approaches, on the other hand, do not estimate the network capacity. They rather observe the current buffer occupancy (i.e., its length in time unit), and request low bitrate segments if the buffer occupancy is low and vice versa.

**Rate-based Algorithms.** FESTIVE [78] ensures that multiple clients, sharing a bottleneck link, converge to a fair bandwidth allocation while achieving high quality and stability across these users.
Figure 2.2: Example of a buffer-rate function used to determine segment bitrate based on current buffer occupancy $b(t)$.

FESTIVE defines a pipeline of three steps to adaptively fetch a video segment: (1) scheduling when the next segment to be downloaded, (2) choosing the bitrate for that segment, and (3) estimating the available bandwidth. Specifically, FESTIVE uses randomized periodic scheduling to calculate when to start fetching the next segment. The randomized scheduling overcomes the biases introduced by initial network conditions and synchronized join time. Second, FESTIVE uses a stateful bitrate selection mechanism, where the rate of increase is a monotonically decreasing function of the current bitrate, and the rate of decrease is constant. Finally, to estimate the bandwidth, FESTIVE uses the harmonic mean of the last 20 bandwidth samples to mitigate bandwidth outliers.

Li et al. [91] observed that off-intervals in video requests lead to ambiguity in knowing the accurate network conditions (e.g., this may result in bandwidth overestimation). They proposed the probe and adapt algorithm (PANDA) to solve such ambiguity, while achieving fairness, high quality, stability and continuous playback (i.e., avoiding rebuffering events). In particular, to estimate the bandwidth, PANDA uses an approach similar to the TCP congestion control. It proactively increases the request bitrate (i.e., probing the network condition) till a congestion is inferred. As a result, the request bitrate backs off accordingly. In addition, PANDA schedules segment download time such that the average data rate is smoothed across the playback interval to reduce the off-intervals.

**Buffer-based Algorithms.** Huang et al. [73] integrated a buffer-based adaptation algorithm in the Netflix player, and they showed that their algorithm reduces the rebuffering events while preserving player video quality and stability. They divided the algorithm into two stages: startup and steady-state. At startup, the buffer is still building up, so they used a rough estimate of the bandwidth to quickly download segments with high quality. At steady-state, they use a buffer-rate function similar to the one in Figure 2.2 for CBR-encoded videos. In particular, the selected bitrate $q(t)$ is calculated using the current buffer occupancy $b(t)$. The x-axis is divided into three regions: lower reservoir, cushion and upper reservoir. The lower reservoir compensates for bandwidth variations.
during $Q_{\text{min}}$. The cushion (the increasing line segment in Figure 2.2) is stretched to reduce rebuffering events and frequent quality switches. Finally, the upper reservoir occupies the largest 10% of the maximum allowed buffer occupancy. For VBR-encoded videos, they replace the buffer-rate function with buffer-chunk one, where the y-axis represents the segment size to be fetched.

Spiteri et al. [121] formulated the rate adaptation problem as a utility maximization problem to achieve high quality and reduce rebuffering events. They modelled the buffer dynamics using queuing theory. Then, they proposed a greedy buffer-based algorithm (called BOLA) that, for every time slot, minimizes the drift in the buffer value that makes the system unstable, while achieving the aforementioned objectives. However, they assume that the number of segments of the video is large in order to absorb the fluctuations in network conditions.

**Hybrid Algorithms.** Hybrid approaches were proposed in the literature to combine both buffer occupancy and capacity estimation in order to improve the overall perceived quality.

Tian and Liu [131] proposed a control-based algorithm that takes buffer occupancy and bandwidth estimation as inputs. The algorithm attempts to be responsive when the bandwidth drops, and smoothly increase the video bitrate when the bandwidth increases. To choose the quality level to be requested, they designed a controller similar to the conventional proportional-integral (PI) controller. The main difference is that the new controller smoothly modifies the requested bitrate, when the system is far from the target buffer (i.e., the equilibrium point). However, this controller does not avoid quality variations after quantizing its output. To achieve stability, they augmented the controller with a stateful switching logic. That is, it upshifts the requested bitrate to the controller output only after number of slots of high buffer occupancy.

Chiariotti et al. [26] assumed that video complexity and network conditions follow Markov processes. Thus, they modelled the rate adaptation problem as a Markov Decision Process (MDP). In particular, they defined set of actions, state space and reward function to maximize perceived quality and stability, and reduce rebuffering events. The goal is to find a policy that maximizes this reward. Since the network conditions are unknown in real-time, the adaptation algorithm incorporates a reinforcement learning process. This ensures that the client learns the probability distribution of state transition for the network conditions. Then, the adaptation policy is calculated using an online state-value update algorithm, and outputs an action that represents the quality of the segment to be fetched.

In order to cope with the network dynamics, Pensieve [97] strives to remove the fixed control of typical rate adaptation heuristics by learning this policy from previous observations. In particular, Pensieve is a reinforcement learning algorithm that trains a neural network as its control policy (called Actor network). At training time, the Pensieve simulator collects observations, and feeds the information about throughput and buffer occupancy to the Actor and Critic networks. Both networks use the policy gradient method to update their parameters. The Actor network produces the policy using an advantage function, which is the difference between the reward of a deterministically-chosen action, and the reward of an action drawn from the policy. The Critic network helps the Actor network estimate the advantage function by calculating the expected total reward given a
certain state. At running time, only the Actor network is needed for making the decision. Pensieve does not assume apriori knowledge of the network condition. It also uses offline learning with richer state space instead of reducing the state space, which yields better results compared to [26].

Yin et al. [148] model rate adaptation objectives as an optimal control problem over a finite period of time in the future. They assume knowing the network conditions during this period. They focused on improving the quality and stability, and reducing rebuffering events and startup delay. Since solving this optimal control problem is computationally extensive for real-time adaptation, the authors propose to solve multiple instances of the problem using the CPLEX solver at the server-side, and the client chooses the quality level by enumerating these pre-calculated solutions.

CS2P [126] builds a bandwidth prediction model and integrates its output with the algorithm proposed in [148]. The model has offline and online components. The offline component aggregates bandwidth measurements from different sessions. For every cluster of sessions sharing the same features, CS2P learns a Hidden Markov Model (HMM) that represents state transition from a current to a future bandwidth value. The hidden states account for factors that affect the bandwidth but are not visible for the client, such as the number of video flows sharing the same link. During the streaming session, CS2P maps the session to the closest HMM, and uses it as a bandwidth predictor.

Wang and Ren [133] proposed a rate adaptation algorithm based on multi-step predictive model. In particular, they noted that selecting the quality level based on an instantaneous measurement of link capacity or buffer occupancy may result in quality fluctuations or rebuffering events. They formulated the cost function of the predictive model as a weighted sum of buffer occupancy differences and quality changes over $K$ steps in the horizon. The optimal solution is to choose the highest video quality lower than the sum of current quality and predicted quality variation. They show that the multi-step predictive model can increase the match between the actual bandwidth and the selected quality levels on the long term. Compared to Yin et al. [148] who assumed perfect knowledge of network conditions, Wang and Ren proposed a control-based capacity estimator that uses Kalman filter to correct the measurement. The main idea is that a good estimator should consider dynamic number of bandwidth samples over time, to balance between the measured download throughput and the predicted capacity (i.e., from the control loop).

QUETRA [146] employs the queuing theory to derive a rate adaptation algorithm to optimize the quality, stability, rebuffering time and startup delay. It models the video buffer as an M/D/1/K queue, which is a queue such that: the video segments arrive the buffer following Poisson distribution, the service rate (i.e., playback rate) is deterministic, there is one server, and the maximum buffer occupancy $K$. Given this model, QUETRA estimates the buffer slack, which is the difference between the maximum buffer occupancy and the average expected buffer occupancy. QUETRA runs as follows. It periodically estimates the network capacity, calculates the buffer slack for the estimated capacity, and chooses the next quality level that minimizes the difference between the expected buffer slack and the current buffer occupancy. This reduces the occurrences of empty buffer that results in rebuffering events, and full buffer that results in off intervals which may overestimate the network prediction [91]. Compared to most adaptation algorithms, QUETRA requires setting
two parameters only: the maximum buffer occupancy and the period length. However, assuming that segment arrival rate follows a Poisson distribution is arguable, and not supported by real measurements.

2.3.2 Immersive Videos

All of the above works address traditional single-view videos, and thus, they cannot handle next-generation videos in which users switch among views or viewports. Rate adaptation algorithms for single-view videos will either waste network bandwidth if they prefetch all views or lead to playback interruptions if they prefetch a single (active) view or viewport only. We categorize these contents to Multiview and Virtual Reality as follows.

Multiview Videos. We categorize multiview videos to three types: 3D Free-viewpoint video [58, 59], 3D Multiview [125], and 2D Multiview [34]. Hamza and Hefeeda [58, 59] proposed an interactive free-viewpoint video streaming system using DASH. The client fetches original views and generates virtual views when required. They proposed a rate-based adaptation algorithm that chooses the quality levels of texture and depth views based on the estimated bandwidth and the rate-distortion model for the target virtual view. This work does not consider the temporal quality variance when choosing different quality levels of original views over time.

Su et al. [125] proposed a rate-based adaptation algorithm for 3D Multiview video streaming using the High Efficiency Video Coding (HEVC) standard. They only address non-interactive 3D videos where users cannot choose the viewing angles. In addition, the adaptation algorithm divides the estimated bandwidth over all views, which may not consider the relative importance for every view.

360-degree Videos. There are two types of VR streaming: monolithic and tile-based streaming. In monolithic streaming, the content provider pre-creates video frames each consisting of multiple viewports, and allocates more bits for popular viewports. Facebook employs such a technique [86] to stream its VR content. In the tiling approach, the video is temporally chopped into video segments (similar to 2D videos). Moreover, every video frame is spatially cut into tiles to save bandwidth. In this type, efficient spatio-temporal adaptation algorithms that choose combinations of tile quality levels over time are required. We note that multiple works [140, 31, 55] have considered calculating the optimal set of tiles by balancing between the coding efficiency and bandwidth waste. In this report, we focus on adaptation algorithms and delivery systems for tile-based streaming.

Xie et al. [143] proposed a buffer-based adaptation algorithm based on head movement prediction. They predict the tile probability, and estimate the total bitrate budget that minimizes the difference between current and target buffer occupancy and avoids rebuffering. They minimize the total expected distortion and the spatial quality variance of tiles given the tile probability distribution and the bitrate budget. They reduce the target buffer occupancy to absorb the prediction errors. They predict the per-tile probability as follows. First, they use linear regression to predict the rotation angles, and assume that the prediction errors follow Gaussian distribution to predict the viewing angle. Second, they predicted the probability of a given spherical point to be viewed by averaging
the viewing angle probabilities. Finally, the probability of a given tile is the average probabilities of the spherical points belong to that tile. The paper assumes linear relationship between the current and predicted angles, and then draws conclusions about the prediction error probability distribution using small dataset of head movements. In addition, the algorithm does not consider the temporal quality variance.

Nasrabadi et al. [102] consider tiled streaming of SVC videos. The server pre-creates base and enhancement layers for every tile in the cubemap projection. The goal is to improve quality while reducing rebuffering and quality switches. To reduce rebuffering, the algorithm aggressively downloads base layers for all tiles up to a threshold buffer occupancy $K$ seconds. If the buffer length exceeds $K$, the algorithm prefetches enhancement layers based on the network throughput (to reduce quality switches), and the prediction of the next viewport using weighted linear regression.

Petrangeli et al. [109] proposed a tiled-based adaptation algorithm based on H.265 encoding, HTTP/2 server push, and estimated bandwidth and future viewport. They use a non-uniform equirectangular projection, where every frame is tiled into six tiles: two polar and four equatorial. The tiles are encoded using H.265 in order to allow the client to decode multiple tiles using one decoder. In addition, to reduce the number of GET requests resulted in requesting multiple tiles, they use HTTP/2 to enable the server to push the requested tiles with a single GET request. The adaptation algorithm estimates the network bandwidth and allocates the lowest quality level for all tiles. It further predicts the future viewport by estimating the fixation speed. Once the current and future viewports are known, the algorithm uses the bandwidth slack to increase the quality level of tiles based on their spatial location to the predicted viewport. That is, the closest tiles to the future viewport are allocated higher quality levels. Similar to [143], this work does not consider quality switches over time.

Alface et al. [115] presented a real-time 16K VR streaming system to mobile untethered clients, where users can zoom in and out in addition to the aforementioned rotation angles. The incentive of this work is that 4K VR streaming often results in pixleated experience, hence, 16K streaming is preferred in such interactive applications. However, 16K streaming poses two challenges. First, the authors conjecture that naive 16K streaming requires bandwidth of 300 Mbps which is a huge overhead for wireless devices and their battery usage. Second, there is no display technology that can render such resolution. They proposed a server-side component that produces two outputs: one 4K stream per user for the active viewport, and one 1K stream shared across all users as a fallback viewport when the corresponding 4K stream is not downloaded due to network latency. To generate the 4K stream, every tile in the scene is independently encoded to 8x8 tiles to allow random access of the files. Then, the tiles of the viewport are chosen and transcoded to one 4K stream to be sent to the user. This work does not consider the spatial relationship between different tiles, which may result in better network utilization. In addition, due to online transcoders, the cost of the system increases as the number of concurrent users increases.
Figure 2.3: Illustration of different interconnection models between ISPs. Customers connect to the Internet through ISPs. Dotted lines correspond to the interdomain traffic between ISPs.

2.4 Content Distribution Networks (CDNs)

In this section, we describe the need for CDNs in video streaming, the challenges of designing CDNs, and collaboration space between CDNs, ISPs, and end users.

The Internet is a network connecting multiple networks (i.e., autonomous systems (ASs) [85]). Specialized ASs, called ISPs, provide Internet access to their users by connecting itself to other ISPs. Figure 2.3 shows an example of network consisting of three ASs: ISP1, ISP2 and ISP3. There are two types flows between ASs [56]. Data flows correspond to the actual data being transferred between them through their physical infrastructure. The second (logical) flow is the cash flow, which represents the monetary relationships between these ISPs. There are two types of cash flows. The first one is peering relationship, where two ISPs agree to carry their traffic for free. Typically, peering links can be connected directly between ISPs (e.g., optical fiber), or through Internet eXchange Point (IXP). The second relationship is customer-provider (i.e., transit) links, where one ISP agrees to carry the traffic of other ISP for money.

To achieve financial gains, ISPs thrive to reduce the amounts of interdomain and intradomain traffic. Reducing interdomain traffic translates to paying less for transit, or keeping peering links less utilized for additional traffic. They also thrive to minimize the maximum link utilization (MLU) inside their networks [119], to optimize the quality of their applications. This is related to the non-linear relationship between link utilization and delay. In particular, Fortz and Thorup [47] modeled the link delay as a piecewise linear function of the link utilization. When link utilization is less than 60%, the link delay is dominated the propagation delay (e.g., the distance between two servers). However, when link utilization approaches 90%, other delays factors such as queuing delays dominate end-to-end link delay.
Meanwhile, the amount of video traffic, carried by ISPs, has been increasing at high rates. In particular, traffic of video applications is expected to occupy 82% of the Internet traffic by 2022 [1]. Since users and servers of content providers (e.g., Netflix) usually exist in geographically distant locations, serving users directly from these servers increases both the interdomain traffic for ISPs, and the latency of receiving video segments for end users. This can stress the ISP network infrastructure as well as reducing the number of its subscribers due to these service delays. As a real-life example, IX Australia [76] reported peering traffic increase up to 100% for some ISPs in Australia [128].

As a result, specialized networks, called content distribution networks (CDNs), are deployed to mitigate these drawbacks by moving content closer to the user. These CDNs lease or deploy servers in the ISP networks to bring multimedia content closer to the user. When a user requests a video that is stored in CDN servers, the service delays and interdomain traffic are reduced. Thus, CDNs are important to improve the quality of experience for end users, and reduce the operational costs for ISPs.

2.4.1 Challenges of CDNs

CDNs are the glue for video streaming applications, that connect users to multimedia content through the ISPs. Thus, their performance affects the operational costs and quality of experience for end users. CDNs attempt to improve their reachability and connectivity to attract more clients. In addition, their infrastructure should be robust to handle the growing demands. We highlight some of the challenges the face traditional CDNs.

Server Selection. Temporal and spatial popularity variation [22, 149, 74] is a major phenomenon in video streaming systems. Specifically, the popularity of videos vary over time and geographical location. Moreover, content providers offer new multimedia content regularly, which results in popularity spikes. Server selection refers to assigning a user request to a server which stores the requested video. Since CDNs attempt to increase their hit rate, server selection plays a major role in CDN performance.

Traffic Engineering. CDN server storage is often limited, and thus user requests and server mis-selection may result in intradomain traffic. Specifically, a video request that does not exist in a nearby CDN server will be directed to a CDN server in a different access network. A major challenge is that most traditional CDNs are network-oblivious. They do not know the current network state of the ISP. As a result, CDNs often combine active network measurements and user feedback to understand the ISP network. However, they are inaccurate and do not guarantee the required quality of service. Moreover, even when CDNs could estimate the network state, it is impossible for CDNs to actually direct the traffic through optimal links.

Cost Minimization. There is a trade-off between cost and performance in the delivery network. CDNs can provide a better quality of service by increasing the storage and network resources, but CDNs often attempt to minimize their costs as well to increase their marginal profits. Also, the idea of shutting down CDNs servers to minimize energy costs [112] was discussed in the literature.
In addition, major CDNs seek to minimize the maintenance, management and upgrade costs [107] through automation.

To summarize, traditional CDNs are not aware of the network state, so they rely on expensive active measurement techniques. This state of network unawareness may result in degrading the performance such as increasing the latency or intradomain and interdomain traffic. In addition, the delivery networks attempts to optimize different contradicting metrics such as costs and latency.

### 2.4.2 Collaboration Space in CDNs

CDNs would collaborate with content providers and users to alleviate the costs of being network-oblivious. Figure 2.4 shows the collaboration space for CDNs. Each point in this space corresponds to a potential collaboration to distribute the content. For example, application-specific CDNs point corresponds to a collaboration between ISPs and content providers. Traditional CDNs point shows that only CDNs are responsible for the content distribution.

**Traditional CDNs.** Akamai [6] and Level3 [87] are examples of traditional CDNs. Although these commercial CDNs share the same objectives and roles, they are different in architecture. Akamai deploys more servers over all the world each with low storage capacity, however, Level3 deploys less servers with more storage capacity.

These CDNs make agreements with ISPs to lease or deploy servers close to users to store multimedia content. Since these storage resources are often limited and content library keeps growing, CDNs store the most popular videos to increase their hit rates. Thus, many algorithms and techniques are devoted to increase the hit rate by replacing the unpopular content with popular one. Specifically, there are two major types of algorithms [119, 90]: pull- and push-based approaches. Pull-based algorithms are the traditional caching algorithms such as LRU [119], where CDN servers
decide what content to be replaced based on content history. On the other hand, a logical centralized controller is usually involved in push-based [90], where this controller tracks popularity variation and pushes content to CDN servers. CDNs usually seeks to maximize the hit rates. Moreover, they usually improve their infrastructure [107], reachability and connectivity to attract more clients and generate profit.

**Broker CDNs.** These brokers attempt to achieve the balance between cost and performance of multiple CDNs serving the same content provider (i.e., content multihoming). This can be important for the following reasons. First, one CDN has different cost and performance at different regions. Second, choosing the best CDN in terms of performance may result in unnecessarily higher costs. Third, in many cases, one CDN at one region may not be able to handle all demands from this region. Last, CDN performance varies over time due to hardware and software failures, or maintenance schedules.

Liu et al. [93] defined the multihomed server selection problem as a minimization of a cost function (e.g., the cost Content Provider pays). This cost function is concave because of the non-linear nature of costs in one CDN. Specifically, most CDNs charge for the next traffic volume less than the current traffic volume. This work distinguished between a passive client who chooses content from one CDN, and an active client who selects content from multiple CDNs. It further provided a polynomial time algorithm for the multihoming server selection problem, and a client-based adaptation to follow the first algorithm decisions.

**ISP-CDN Collaboration.** As we discussed in Section 2.4.1, most traditional CDNs are not aware of the current network state, which results in higher costs and lower performance. To overcome these problems, ISP can share with CDN information about its resources.

Frank et al. [49] implemented a prototype of NetPaaS, a system that enables ISP and CDN collaboration in real-time. NetPaaS provides three main interfaces (1) user-server mapping: CDN asks ISP for recommendations (e.g., a list of sorted servers) to assign users, (2) resource sharing: ISP shares location, capacity and number of available storage servers as well as their pricing, and (3) content-server mapping: CDN asks ISP to scale up or down the servers to allocate the content.

We note two issues in NetPaaS. First, the CDN is still network-oblivious, but sharing information about storage resources may help balancing performance and cost. Second, NetPaaS may not be the best solution to manage the ISP network (e.g., the interdomain and intradomain traffic flows).

**ISP-P2P Collaboration.** P2P [30] can augment the video streaming system by fetching multimedia content from nearby users. However, pure P2P systems, similar to CDNs, are network-oblivious. They do not know the current characteristics of the underlying ISP network. So they may result in network imbalances and unfairness, and performance degradation of the streaming sessions.

Xie et al. presented P4P [142], a system where ISP shares information with applications (e.g., peers) to overcome pure P2P problems. P4P consists of two components: iTracker deployed into the ISP, and appTracker deployed at peers. iTracker shares information about its network such as distance (e.g., OSPF weights), capability (e.g., bandwidth) and policy (e.g., for shape traffic during peak hours). P4P attempts to minimize MLU. Specifically, peers use this information to choose
other peers and their traffic to minimize MLU. Authors showed that the optimization problem can be solved independently by applications without a centralized controller.

**CDN-P2P Collaboration.** Many works [4, 67, 68, 12] discussed hybrid CDN-P2P delivery systems. We focus on the most recent one in this section.

There has been a thought that P2P works well for live streaming only, due to the synchronous nature of live streaming (i.e., many users watch the same content are able to share it). Balachandran et al. [12] revisited this thought by characterizing the potential benefits of P2P-augmented CDNs after studying user behavior of VoD system. Specifically, they studied VoD of 4 million users and 14 million streaming sessions across two months. They found that P2P helps reducing CDN accesses up to 87% due to synchronous viewing during peak hours. Also, P2P can be used in the beginning of the streaming sessions (e.g., before P2P stabilizes) if content provider isolates users who quit early.

**Application-specific CDNs.** Major content providers, such as Netflix, affect ISP network by increasing peering traffic. For example, IX Australia [76] reported peering traffic increase up to 100% for some ISPs in Australia [128].

Application-specific CDNs are specialized delivery networks created to handle one content provider. To accommodate the demand growth, Netflix deploys its own delivery network called OpenConnect [103]. Specifically, Netflix provides two options for ISPs. First, peering links between ISPs and IXP connected to Netflix servers. Second, Netflix deploys its own cache servers inside the ISP network, these servers are called OpenConnect Appliance (OCA). Also, Netflix provides public and private peering locations [108] for the two options. OCAs deployed into a specific region store multimedia content according to its popularity in that region. Netflix adopts a push-based approach to update OCAs during off-peak hours within a 12-hour window.

**Telco-CDNs.** Recently, major ISPs such as AT&T start managing and deploying CDNs inside their networks [75]. Unlike traditional CDNs, these emerging delivery networks, called telco-CDNs, enable ISPs to manage the network resources as well as server resources of the delivery network. Thus, telco-CDNs select servers based on content popularity, server capacity, and interdomain and intradomain traffic flows.

Li and Simon [90] adopted a push-based approach to manage telco-CDNs. They formulate the problem to minimize the costs of assignment (e.g., content-server mapping) and service (e.g., user-server mapping). They showed that the Capacitated Facility Location Problem can be reduced to their problem, which is in NP-complete. To overcome the hardness of the problem, they proposed a genetic algorithm and implemented it in MapReduce.

Zhou et al. [152], similar to [90], proposed an algorithm to manage telco-CDNs. However, their goal was different, they sought to maximize the number of live streaming sessions with lowest bandwidth in telco-CDN. They also showed that managing telco-CDN jointly outperforms two-step processing up to 10% in the number of delivered live streaming sessions.
Multicast Systems

Multicast is a group communication pattern where multiple destinations of a single session receive the same data. To reduce the network usage, multicast has been used in various applications such as IPTV [54], video conferencing [24] and massive multiplayer games [27]. Moreover, recent large-scale Internet applications such as Facebook Live [136, 114] introduced a renewed interest in multicast services.

At a high level, there are two approaches to implement multicast: application-layer and network-based multicast. In application-layer multicast, the hosts implement the multicast functions such as forwarding packets and maintaining sessions. In contrast, network-based multicast systems require the routers to forward and duplicate packets of multicast sessions. We describe both approaches in the following.

2.5.1 Application-layer Multicast

As depicted in Figure 2.5a, hosts in application-layer multicast (ALM) systems are responsible of sending the data of a session to other hosts in that session. In other words, the hosts form an overlay network to forward and duplicate packets to the destinations. Although ALM systems are easy to deploy, they lack the knowledge of the underlying network topology and conditions, and they cannot achieve TE objectives. As a result, they may impose additional link overheads and service delays because a packet may be transmitted on the same link multiple times.

One of the main tasks of designing efficient ALM systems is to construct and maintain the overlay multicast tree. There are two approaches to achieve this task [65]: mesh-based and tree-based algorithms. We discuss the two approaches in the following.

Mesh-based Approaches. In these systems [66, 100, 145, 150], the hosts first construct and maintain a mesh topology between them, then the multicast tree is computed using that mesh topology. Specifically, the source of a multicast session is chosen as the root of the tree, and the paths of the
tree are calculated by running a routing algorithm in the mesh (relative to the source). Notice that the performance of the multicast tree depends on the mesh topology.

Narada [66] performs latency measurements to dynamically optimize the mesh topology by adding and dropping links from that mesh. To achieve robustness, each host in Narada needs to maintain state about all other hosts. Thus, Narada is suitable for small-scale sessions because it imposes control overheads especially when a host joins/leaves the mesh. Scribe [100] is built on top of Pastry [116] routing and lookup algorithms to build large-scale multicast trees. CoolStreaming [145] builds a mesh topology where every host maintains a subset of the active neighbors and their status. CoolStreaming uses a randomized gossip-based algorithm to refresh its data structures. Moreover, there are no predefined roles for the hosts (e.g., parent/child). Instead, data is forwarded by hosts based on its availability.

**Tree-based Approaches.** Tree-based ALM systems [48, 15, 13, 132, 77] build multicast trees directly without the need for a mesh topology. Each host in a multicast session needs to choose a parent from a set of other hosts in the network. This is limited by the maximum number of children a parent can have. Moreover, these systems may need to execute auxiliary algorithms such as loop detection and avoidance to guarantee efficient forwarding.

The tree-based systems impose lower communication overheads compared to mesh-based approaches. In addition, these systems give direct control over constructing the multicast tree. However, tree-based systems have a single point of failure, which happens when a host crashes or leaves ungracefully. In this case, all of that host's children may stay disconnected for a period of time. Mesh-based systems are more robust and resilient against failures since they do not have a single point of failure. This is because a new host can be chosen from the predefined mesh topology. In addition, they are more suitable for multi-source multicast applications. However, they often impose significant control overheads.

### 2.5.2 Network-based Multicast

As mentioned earlier, application-layer multicast systems are easy to deploy. However, they cannot be used to implement TE objectives. In addition, they may result in additional link overheads. In contrast, the routers in network-based multicast systems forward and duplicate the data as shown in Figure 2.5b. Specifically, nodes in the distribution tree represent routers, not end users. The root of the multicast tree is referred to as the ingress router, where the multicast traffic of the session is injected. Leaf nodes in the multicast tree are referred to as egress routers, where end users of the multicast session are attached. To join or leave a session, end users notify the corresponding egress router. Edges of the distribution tree may not follow shortest paths computed by intra-domain routing protocols. That is, paths in the tree are chosen to satisfy the objectives of the ISP administrator (e.g., minimize the maximum link utilization).

Efficient network-based multicast forwarding systems need to address three challenges (i) scalability, (ii) efficiency, and (iii) generality. The scalability refers to the amount of state to be maintained at routers and the cost of updating this state. Efficiency of a multicast forwarding system
means that it should not result in loops or additional traffic. Finally, the generality of a multicast system indicates that it can support different tree sizes, traffic patterns, and network topologies.

We divide existing network-based multicast forwarding systems into rule-based and label-based systems. Rule-based systems maintain the forwarding state (i.e., rule) about each session at routers. Label-based systems move the forwarding information to labels attached to packets.

**Rule-based Approaches.** These multicast approaches require storing forwarding state about sessions at routers. The traditional IP multicast [33] is an example of such approaches. IP multicast is implemented in core routers produced by most vendors. However, it suffers from scalability issues in real deployments [38]. In particular, group management and tree construction protocols (e.g., IGMP [19] and PIM [45]) require maintaining state at routers for each session, and they generate control messages among routers to refresh and update this state. Maintaining and refreshing state for each session impose significant overheads especially on core routers that need to support high-speed links. Thus, in practice, router manufacturers tend to limit the number of multicast sessions. For example, the Cisco ASR 9000 series can maintain up to 1,024 multicast sessions [2]. In addition, IP multicast uses shortest paths and cannot implement traffic-engineered trees.

Recent SDN-based protocols isolate the control plane from the data plane, and can implement traffic-engineered paths. For example, a rule-based approach can be implemented using OpenFlow [82], where the controller installs header-matching rules and actions to forward/duplicate packets. Since OpenFlow stores state at every router along the multicast trees, the total forwarding state at routers grow with the number of sessions. Moreover, the controller has to schedule rule updates when trees change over time in order to guarantee efficient forwarding [80, 151]. Li et al. [89] proposed a multi-class Bloom filter (MBF) to support multicast in data center networks. For every interface in a router, MBF uses a Bloom filter to store whether packets of a session should be duplicated on that interface. MBF cannot completely avoid redundant traffic due to the probabilistic nature of Bloom filters.

**Label-based Approaches.** These approaches move the forwarding information (completely or partially) to labels on packets. For example, mLDP [130] enables multicast in label switched paths (LSPs). However, mLDP forwards traffic on the shortest paths and cannot support traffic-engineered trees. Moreover, it requires an additional protocol to distribute labels among routers. BIER [135] encodes global router IDs of tree receivers in the label as a bit map. Similar to mLDP, BIER only supports shortest paths. LIPSIN [81] uses a Bloom filter as label to encode global link IDs of a tree in publish/subscribe applications. LIPSIN may result in redundant traffic or forwarding loops because of the probabilistic nature of Bloom filters. Thus, LIPSIN requires an additional protocol where downstream routers notify upstream ones if they falsely receive a packet. This protocol imposes additional state and communication overheads on routers.

Segment routing (SR) [46] is a recent proposal to support traffic-engineered unicast flows. It was later extended to support multicast by considering every tree as a segment in the network. It attaches one label containing the tree ID to packets of a session. Routers along the tree maintain a mapping between that label and the output interfaces (i.e., forwarding state). SR and its extensions
do not need label distribution protocols such as LDP [10]. However, SR multicast extensions require maintaining state at routers for every tree.

In this thesis, we address the challenges of network-based multicast systems because they support implementing traffic-engineered multicast trees in the data plane. We propose two network-based multicast forwarding systems to address different ISP network environments. We present a multicast forwarding system called STEM to implement traffic-engineered trees in ISP networks where the controller cannot update the forwarding rules at core routers. In addition, we address network environments that allow the controller to update the state at core routers. We describe the design of a multicast forwarding system, called Helix, that splits information about trees into a label and state maintained at some routers.
Chapter 3

MASH: A Rate Adaptation Algorithm for Multiview Video Streaming over HTTP

In this chapter, we present the design and analysis of a novel client-based multiview adaptive streaming over HTTP algorithm. Then, we assess its performance through actual implementation and comparisons against the closest algorithm in the industry.

3.1 Introduction

Recently, there has been significant interest in streaming immersive multimedia content such as multiview and virtual reality (VR) videos. These videos consist of multiple views of the same scene but captured by several cameras at different angles. For example, YouTube [3] released a small-scale experiment to stream multiview content in early 2015, in which a user can experience the scene (a concert in that experiment) from different perspectives by switching among various views. Two of the main challenges in streaming this complex multiview video content over the Internet are adaptation to dynamic network conditions and supporting user interactivity; which we address in this chapter.

Current popular streaming services such as YouTube and Netflix employ adaptive streaming over HTTP, using standards such as DASH [123]. In such services, the video is encoded into multiple quality levels at different bitrates. Each quality level is divided into segments of equal length. During the streaming session, clients request segments from the HTTP server. To handle varying network conditions, a rate adaptation algorithm is used by each client to dynamically request segments with the most suitable quality for the current conditions. The rate adaptation algorithm needs to balance multiple, sometimes conflicting, performance metrics including quality level, quality variations, and smoothness of playback [39, 91].

Unlike single-view videos, multiview videos require much more complex rate adaptation algorithms. This is because these algorithms need to fetch segments for active views as well as other views to enable the user to smoothly navigate across views. In addition, the video may have many
possible views, and naively fetching segments from all of them can lead to significant waste in network bandwidth. On the other hand, fetching segments from only the active view and nothing from other views will introduce long delays when a user switches to another view, which can damage the immersive experience and lead to user dissatisfaction. Furthermore, users interact with multiview videos in different ways, based on their own interests and perspectives. Thus, predicting the view that may be needed next is not straightforward. These complications are all added to the challenge of handling the network dynamics. Although the rate adaptation problem for single-view videos has been extensively studied in the literature [73, 148, 91, 7], it has received little attention for the more complex multiview videos, which are expected to be quite popular in the near future given the huge interest and investments of major companies such as Google, Facebook, Microsoft, and Samsung.

In this chapter, we propose a novel Multiview Adaptive Streaming over HTTP (MASH) algorithm. MASH introduces a new perspective to the rate adaptation problem in multiview video streaming systems: it constructs probabilistic view switching models and it utilizes these models in dynamically selecting segments of various views at different qualities, such that the quality and immersiveness of the videos are maximized while the network bandwidth is not wasted. Specifically, for each multiview video, MASH constructs global and local switching models. The global model captures user activities across all streaming sessions seen by the server so far, while the local model understands user activity during the current streaming session only. MASH combines the two models to dynamically weigh the importance of each view given the one being watched. Then MASH uses a new buffer-based approach to select segments of different views according to their relative importance. The contributions of this chapter are summarized as follows:

- We present view switching models based on Markov chains to capture user activities. We combine these models dynamically to prioritize the views (Section 3.3).
- We propose a new buffer-based rate adaptation algorithm for multiview video streaming systems that use the HTTP protocol. We show that the proposed algorithm and view switching models impose negligible overheads and we analyze their convergence properties (Section 3.5).
- We developed a multiview video player and implemented our rate adaptation algorithm and view switching models in it. We conducted extensive empirical study to compare our algorithm versus the one used by YouTube for multiview videos, and the results show substantial improvements across multiple performance metrics and in different network conditions. For example, MASH achieves up to 300% improvement in the average rendering quality and up to 200% improvement in the prefetching efficiency compared to the YouTube algorithm. In addition, our results show that MASH achieves fairness across concurrent sessions, and it does not overload the streaming server. Furthermore, we compare MASH versus other rate adaptation algorithms, which are derived from current rate adaptation algorithms for single-view videos, and we show that MASH outperforms all of them (Section 3.7).
The chapter is organized as follows. We summarize the related works in Section 3.2. Then, we present an overview of our approach and describe the proposed view switching models in Section 3.3. We present the details of the proposed MASH rate adaptation algorithm in Section 3.5. We present our empirical evaluation and comparison in Section 3.7, and we conclude the chapter in Section 3.8.

### 3.2 Related Work

We categorize rate adaptation algorithms to rate-based [78, 91, 101, 131], buffer-based [73] and hybrid [148] approaches. In rate-based approaches, the algorithm estimates the network capacity and chooses suitable segments accordingly. For example, FESTIVE [78] uses the harmonic mean to smooth the measured network capacity. It also uses randomized segment scheduling to mitigate differences in users join time. The probe and adapt algorithm [91] smoothes the estimated bandwidth using an approach similar to the TCP congestion control, and schedules segment requests to avoid buffer underflows. Accurate network capacity estimation is, however, difficult [72]. Buffer-based approaches, on the other hand, do not estimate the network capacity. They rather observe the current buffer occupancy (i.e., its length in time unit), and request low bitrate segments if the buffer occupancy is low and vice versa. For example, Huang et al. [73] integrated a buffer-based adaptation algorithm in a Netflix player, and they showed that their algorithm reduces the rebuffering events while preserving player video quality and stability. A hybrid approach [148] was proposed in the literature to combine both buffer occupancy and capacity estimation to improve the overall quality. This approach models rate adaptation as an optimal control problem, and solves it by enumerating pre-calculated solutions at the client-side. All of the above works address traditional single-view videos, and thus, they cannot handle multiview videos in which users switch among views. Rate adaptation algorithms for single-view videos will either waste network bandwidth if they prefetch all views or lead to playback interruptions if they prefetch the active view only.

Few recent works considered more complex videos, but none addressed our problem of rate adaptation for multiview videos, to the best of our knowledge. For example, Hamza and Hefeeda [58] proposed an interactive free-viewpoint video streaming system using DASH. The client fetches original views and generates virtual views at the client side when required. Their work, however, focuses on optimizing the quality of the synthesized virtual views using rate-distortion models. Su et al. [125] proposed a rate adaptation algorithm for 3D video streaming using the High Efficiency Video Coding (HEVC) standard. Unlike our work, they only address non-interactive 3D videos where users cannot choose the viewing angles. Another set of works address 3D tele-immersive systems (3DTI). 3DTI systems have different interactivity semantics and streaming models than our work. Such systems are multi-user, real-time, and object-oriented, while our work is concerned with video on demand streaming with view-level interaction by individual users. Thus, 3DTI systems address different problems such as object-level [139], frame-level [147] and perception-based [138] adaptation.
The closest approach to our work is the multiview player of YouTube, which is integrated in
the Google Chrome web browser. It allows users to navigate to different views and it dynamically fetches segments of different views during the streaming session. We compare MASH against
the algorithm used by YouTube. Furthermore, we modify recent buffer-based rate adaptation algo-
rithms [73] to support multiview videos, and we compare against them.

3.3 Overview

We consider an adaptive streaming system based on HTTP using standards such as DASH [123],
where a server streams multiview videos to diverse users over dynamic networks. A multiview video consists of $N$ views, which enables users to experience the video from different angles. A user is allowed to interact with the video to choose the desired view at anytime. Supporting multiview videos is, however, challenging. First, the streaming system needs to support efficient view switching, without imposing long waiting delays (when switching happens, which could damage the viewing experience) or consuming excessive network resources (by naively sending all views to every user, even if most of them will not likely be viewed). Second, the streaming system needs to adapt to the dynamic network conditions of different clients and serve the best possible quality in such conditions. The proposed MASH algorithm addresses these challenges, by constructing probabilistic view switching models and using these models in a novel buffer-based rate adaptation method.

Each view $V_i$ is encoded into multiple quality levels $Q_i \in \{Q_{min}, \ldots, Q_{max}\}$ (measured in Mbps). Each quality level is divided into equal-length segments and stored at the server. At the client side, MASH decides which segments of which view(s) and at what quality levels should be requested from the server. When requested segments of view $V_i$ arrive at the client, they are decoded and stored in the frame buffer corresponding to that view. We use $B_i(t)$ (in seconds) to denote the buffer occupancy of the frame buffer of view $V_i$ at time $t$. We refer to the view being watched as the active view, while others are called inactive views. When a user switches to view $V_j$ and it happened that the corresponding buffer is empty, the client will not be able to render $V_j$ and playback interruption (or re-buffering) occurs. To minimize these re-buffering events, we design a composite model for predicting view switching, which captures the switching pattern of: (i) all users who watched this multiview video before (global model), and (ii) the user of the current streaming session (local model).

Figure 3.1 shows a high-level overview of MASH, which runs at the client side. MASH combines the outputs of the global and local view switching models to produce a relative importance factor $\beta_i$ for each view $V_i$. MASH also constructs a buffer-rate function $f_i$ for each view $V_i$, which maps the current buffer occupancy to the segment quality to be requested. The buffer-rate functions are dynamically updated during the session; whenever a view switch happens. As shown later, MASH strives to produce smooth and high quality playback for all views, while not wasting bandwidth by carefully prefetching views that will likely be watched.
3.4 View Switching Models

MASH combines the outputs of two stochastic models (local and global) to estimate the likelihood of different views being watched. We define each view switching model as a discrete-time Markov chain (DTMC) with \( N \) (number of views) states. An illustrative example is shown in Figure 3.2a for a video with four views. View switching is allowed at discrete time steps of length \( \Delta \). The time step \( \Delta \) is the physical constraint on how fast the user can interact with the video. For example, we do not expect the user to switch views more than once in less than 100 ms, because of the limitations on the input interface (e.g., touch screen). In the following we describe the construction of each model. Then, we describe how we combine the two models together. We note that the construction of the switching models is carefully done to ensure convergence as well as minimize the imposed overheads on the system, as detailed in Section 3.6.

**Local Model.** It captures the user activities during the current streaming session, and it evolves with time. That is, the model is dynamic and is updated with every view switching event that happens in the session. The model maintains a count matrix \( M(t) \) of size \( N \times N \), where \( M_{ij}(t) \) is proportional to the number of times the user switched from view \( V_i \) to \( V_j \), from the beginning of the session up to time \( t \). The count matrix \( M(t) \) is initialized to all ones. Whenever a view switching occurs, the corresponding element in \( M(t) \) is incremented. Increasing that element by 1, however, may result in wide fluctuations in the switching model, especially early in the session where all elements still have small values (typically 1). We use an exponential weighted moving average (EWMA) to smooth out these fluctuations. Thus, after switching from \( V_i \) to \( V_j \), we set \( M_{ij}(t) = \gamma M_{ij}(t - \Delta) + (1 - \gamma)(M_{ij}(t - \Delta) + 1) \), where \( \gamma \) is a smoothing factor. In our experiments, we set \( \gamma = 0.2 \). The count matrix \( M(t) \) is used to compute the probability transition matrix of the local model \( L(t) \), where \( L_{ij}(t) = M_{ij}(t) / \sum_k M_{ik}(t) \). A row of \( L(t) \) is denoted by \( L_i(t) \), and it is a vector of size \( N \) that represents the conditional probability distribution \( p(V_j|V_i) \) for every \( j \) (e.g., the probabilities in
Figure 3.2: (a) Simple example of the stochastic switching model used by MASH. (b) The sigmoid function used in assigning importance factors to different views.

Figure 3.2a). At the end of the streaming session, the final transition matrix $L(t)$ is sent to the server to update the global model.

**Global Model.** This model aggregates users activities across all streaming sessions that have been served by the server so far. At beginning of the streaming session, the client downloads the global model parameters from the server. We use $G$ to denote the transition matrix of the global model, where $G_{ij} = p(V_j|V_i)$ is the probability of switching to $V_j$ given $V_i$. If this is the fist streaming session, $G_{ij}$ is initialized to $1/N$ for every $i$ and $j$.

**Combined View Switching Model.** The local and global model complement each other in predicting the (complex) switching behavior of users during watching multiview videos. For example, in some streaming sessions, the user activity may significantly deviate from the global model expectations, because the user is exploring the video from different viewing angles than most previous users have. Or the multiview video may be new, and the global model has not captured the expected view switching pattern yet. On the other hand, the local model may not be very helpful when the user has not had enough view switches yet, e.g., at the beginning of a streaming session. We combine the local and global models to compute an importance factor $\beta_i$ for each view $V_i$ as follows. We normalize the elements of the local and global transition matrices by subtracting the diagonal elements from them; for $L(t)$ (and similarly for $G$):

$$\bar{L}_{ii}(t) = 0,$$
$$\bar{L}_{ij}(t) = L_{ij}(t)/(1 - L_{ii}(t)), \forall j \neq i, 1 \leq i, j \leq N. \quad (3.2)$$

These distributions are similar to the original ones with initializing the diagonal elements by zeros, and updating the counting matrix only when there is a view switch. Given the active view is $V_i$, we calculate its importance factor $\beta_i$ as well as the importance factors $\beta_j$ of all other views $V_j \forall j \neq i$.
at time $t$ as follows:

\[
\begin{align*}
\beta_i & = 1, \\
\beta_j & = \alpha_i \times \tilde{L}_{ij}(t) + (1 - \alpha_i) \times \tilde{G}_{ij}, \forall j \neq i, \\
\alpha_i & = h\left(\frac{1}{N-1} \sum_{j=1, j \neq i}^N \left(\tilde{L}_{ij}(t) - \tilde{G}_{ij}\right)^2\right), \\
E(\tilde{L}_i(t), \tilde{G}_i) & = \frac{1}{\sqrt{2}} \sqrt{\sum_{j=1, j \neq i}^N \left(\tilde{L}_{ij}(t) - \tilde{G}_{ij}\right)^2} \\
h(x) & = (1 + \exp(-(ax-b)))^{-1}
\end{align*}
\]  

Equation (3.3) states that the active view is the most important view. We calculate the inactive view importance factor $\beta_j$ as a linear combination of $\tilde{G}_i$ and $\tilde{L}_i(t)$ in Equation (3.4), where the $\alpha_i$ parameter is carefully computed (by Equations (3.5)–(3.7)) to dynamically adjust the relative weights of the global and local models. Specifically, $\alpha_i$ is calculated as a normalized root mean squared error $E(\cdot)$ of $\tilde{G}_i$ and $\tilde{L}_i(t)$ in Equation (3.6). The denominator $\sqrt{2}$ is the maximum value, since both $\tilde{G}_i$ and $\tilde{L}_i(t)$ have values less than 1. The intuition behind $E(\cdot)$ is to prioritize only the most important inactive views. At the beginning of a streaming session, $E(\cdot)$ is expected to be high, thus, all inactive views will have low $\beta$ values unless they are very important in the global model. During the streaming session, if the user activity approaches the global model, then $E(\cdot)$ decreases and the global model output would have a higher weight. If the user activity deviates from the global model, $E(\cdot)$ is high and $\beta$ will follow the local model more. After computing $E(\cdot)$, we apply a shifted sigmoid function $h(\cdot)$ as in Figure 3.2b. We design $h(\cdot)$ such that its domain and range are in the period $[0, 1]$, and it grows faster than the identity function when $E(\cdot)$ increases. The design of $h(E(\cdot))$ ensures that the local model weighs more, unless an inactive view is very important in the global model. Thus, MASH fetches only important segments to save bandwidth while ensuring smooth playback.

### 3.5 Details of MASH

The proposed combined view switching model in the previous section results in an importance factor $\beta_i$ for each view $V_i$, which is computed dynamically at each view change. Using the importance factors for all views, MASH determines the quality for the segments to be requested.

MASH is a buffer-based rate adaptation algorithm for multiview videos, which means it determines the requested segment quality based on the buffer occupancy level, and it does not need to estimate the network capacity. Previous buffer-based algorithms [73, 148] are designed for traditional single-view videos. Thus, they employ a simple buffer-rate mapping function. This function basically defines two parameters: the minimum $B_{min}$ and maximum $B_{max}$ buffer occupancies in seconds. If the buffer level is less than $B_{min}$, the requested quality for the segment is set to the minimum quality $Q_{min}$. And if the buffer level is greater than $Q_{max}$, the requested quality is set...
to the maximum quality $Q_{\text{max}}$. For buffer levels between $Q_{\text{min}}$ and $Q_{\text{max}}$, the requested quality is linearly proportional to the slope of the function: $(Q_{\text{max}} - Q_{\text{min}}) / (B_{\text{max}} - B_{\text{min}})$.

Rate adaptation for multiview videos is far more complex, as it needs to handle many views of different importance, while not wasting network bandwidth or resulting in many stalls during playback for re-buffering. To handle this complexity, we propose employing a family of buffer-rate functions, which considers the relative importance of the active and inactive views and how this relative view importance dynamically changes during the streaming session. Specifically, we define a function $f_i(B_i(t))$ for each view $V_i$, which maps the buffer level $B_i(t)$ of that view to a target quality $Q_i(t)$ based on its importance factor $\beta_i$ at time $t$. We use $\beta_i$ to limit the maximum buffer occupancy level for view $V_i$ as: $B_{\text{max},i} = \beta_i \times B_{\text{max}}$. Since we set $\beta_i = 1$ for the active view, the algorithm can request segments up to the maximum quality $Q_{\text{max},i}$. Notice that this equation implies that $\beta_i \leq \beta_j \iff B_{\text{max},i} \leq B_{\text{max},j} \iff Q_{\text{max},i} \leq Q_{\text{max},j}$. Thus, for inactive views, MASH can request segments for up to a fraction of their maximum qualities. Figure 3.3 illustrates the buffer-rate functions for two views $V_i$ and $V_j$. $V_i$ is the active view, so $B_{\text{max},i} = B_{\text{max}}$. The figure shows when the requests stop for both $V_i$ and $V_j$, and the maximum bitrate difference to reflect the importance of each view.

We show the pseudo code of MASH in Algorithm 1. We first note that there is an initialization phase of MASH (not shown in Algorithm 1), which occurs in the beginning of the streaming session. In this phase, MASH downloads the global transition matrix $G$ from the server, and calculates the local model transition matrix $L(0)$ and its normalized form $\bar{L}(0)$ using Equations (3.1) and (3.2). Then, given an initial active view $V_i$, the view importance factors $\beta_k$ (Equations (3.3) and (3.4)) and $B_{\text{max},k}$ are calculated for all $k \neq i$. Finally, the buffer-rate functions $f_k$ are calculated given $B_{\text{min}}$, $B_{\text{max},k}$ and the available quality levels. After the initialization phase, MASH runs periodically
Algorithm 1 MASH rate adaptation algorithm.

Require: \( V_i, V_j \): current and previous active views.
Ensure: \( \text{requests} \): segments to be requested from server.

1: \( M_{ji}(t) = \gamma M_{ji}(t - \Delta) + (1 - \gamma)(M_{ji}(t - \Delta) + 1) \)
2: \( L_{ij}(t) = M_{ij}(t)/\sum_{k}^{N} M_{jk}(t), \forall 1 \leq l \leq N \)
3: if \( (V_i \neq V_j) \) then
4: \( \bar{L}_{ii}(t) = 0; \bar{L}_{ik}(t) = L_{ik}(t)/(1 - L_{ii}(t)), \forall k \neq i \)
5: \( \alpha_i = h(E(\bar{L}_i(t), \bar{G}_i)) \)
6: for \( (V_k \in V) \) do
7: \( \beta_k = 1; B_{max,k} = B_{max} \)
8: else
9: \( \beta_k = \alpha_i \times \bar{L}_{ik}(t) + (1 - \alpha_i) \times \bar{G}_{ik} \)
10: \( B_{max,k} = \beta_k \times B_{max} \)
11: end if
12: \( f_k = f(B_{min}, B_{max,k}, Q_{min,k}, Q_{max,k}) \)
13: end for
14: end if
15: for \( (V_k \in V) \) do
16: \( \text{requests.add(getRequest}(f_k, B_k(t))) \)
17: end for
18: return \( \text{requests} \)

every \( \Delta \) seconds and it updates the local counting and transition matrices given \( M(t - \Delta) \). It then checks if a view switch happens since the last invocation (Line 3), and updates its state accordingly. First, MASH updates normalized local vector \( \bar{L}_i(t) \) for the current active view. Second, it updates parameter \( \alpha_i \) for the current active view \( V_i \) (Line 5). Third, MASH iterates over all views and updates their importance factors \( \beta_k \) and maximum buffer occupancy \( B_{max,k} \) values. Then, it updates the buffer-rate functions according to these new values (Line 13). Finally, at Line 17, MASH uses corresponding buffer-rate function \( f_k \) to compute segments rates.

3.6 Analysis and Overheads of MASH

We first show the proposed algorithm does not impose and significant overheads, neither on the client nor on the server. Then, we analyze the properties of the view switching models.

Overheads. MASH requires two stochastic matrices: \( G \) and \( L \), each of size \( N^2 \), where \( N \) is the number of views. For simplicity, let us assume that each element in all matrices takes 2 bytes, although more optimization is possible. Since \( G \) is maintained at the server, the client would need to download extra \( 2N^2 \) bytes at the beginning of the streaming session. At the end of the session, the client uploads the final local model \( L \) to the server to update \( G \), which also takes \( 2N^2 \) bytes. The client stores \( G \) and \( L \) in the local memory, along with the counting matrix \( M \) used to simplify calculation of \( L \). Thus, the added memory requirement is \( 6N^2 \) bytes. On the server side, \( G \) is updated only once after the completion of each streaming session with total computational complexity of
$O(N^2)$. On the client side, the initialization of $L$ and $M$ matrices takes $O(N^2)$ steps, but it is done once. During the session, only $O(N)$ steps are required to update $M$ and $L$ with every view switch. To put these bounds in perspective, consider a rich multiview video with 100 views. The client will download/upload up to 20 KB and need up to 60 KB of memory. These values are insignificant compared to the size of the multiview video (100s of MB). To create and maintain the matrices, only $N^2 = 10K$ simple operations are performed in few msec per each session (which lasts for minutes).

**Analysis of the View Switching Models.** We focus on two properties in the context of multiview video streaming: (1) convergence and (2) rate of convergence. These properties imply whether and when such models represent user activities.

**Lemma 1 (Model Convergence).** The global and local view switching models converge to the stationary distributions $\pi^{(g)}$ and $\pi^{(l)}$, respectively.

**Proof.** We constructed our models with the following properties, which reflect real multiview video applications. First, each model has a finite state space with state count equals $N$, which represents the finite views of the video. Second, the user can switch from any view to any view, that is, any state $i$ can reach any other state $j$. Thus, our models are irreducible. Third, each state $i$ can return to the same state $i$ in irregular number of view switches. Hence, our models are aperiodic. It was shown in [106] (Theorem 1.8.3) that a finite state, irreducible, and aperiodic Markov chain converges. That is, for our global $G$ (and similarly local $L$), there exists a unique probability distribution $\pi^{(g)}$ such that $\pi^{(g)} = \pi^{(g)} G$. That is, the global and local models converge to $\pi^{(g)}$ and $\pi^{(l)}$, respectively. $\square$

We note that by definition, $\pi^{(g)}$ is the left eigenvector $\vec{e}$ of $G$ where its corresponding eigenvalue $\lambda_1 = 1$, and it equals to $\vec{e}/ \sum_i N e_i$. Such eigenvalue $\lambda_1 = 1$ exists since $G$ is a stochastic matrix. Moreover, there are $N$ eigenvalues such that $\lambda_1 > |\lambda_2| \geq \cdots \geq |\lambda_N|$. Given the eigenvalues $\lambda$’s in the above lemma, the global model $G$ can be shown to converge to $\pi^{(g)}$ in $O(|\lambda_2|^k)$ steps. Specifically, it was shown in [28] (Theorem 5.1, p. 378) that the convergence rate of such model is exponential in the order of the second dominating eigenvalue $\lambda_2$. Since the global model is updated at the end of a streaming session, $k$ is the streaming sessions count. We note that as $\lambda_2$ approaches 0, the model convergences very quickly (i.e., requires a small number of streaming sessions). $\lambda_2$ is small when most users agreed on watching specific views. As $\lambda_2$ increases, the global model needs more streaming sessions to reach its stationary distribution. This fact shows the advantage of the local model to capture user activity during single streaming session, where such user may deviate from people consensus, or there are no enough streaming sessions yet to represent such agreement.

### 3.7 Evaluation

We assess the performance of MASH through actual implementation and comparisons against the state-of-the-art algorithm in the industry. We also analyze the fairness of our algorithm across con-
current multiview video sessions and compare its performance against two variations of current rate-adaptation algorithms that could be used for multiview videos.

### 3.7.1 Experimental Setup

We have implemented a complete multiview video player in about 4,000 lines of Java code. It consists of HTTP client, decoders, renderer and rate adapter. Each view has its own decoder and frame buffer. The rate adapter decides on the segments to be requested and their quality levels. Then, segments are fetched from the HTTP server. Once a segment is fetched, it is decoded to frames and stored in the corresponding frame buffer. The renderer has references to all frame buffers, and it renders the corresponding active view.

Figure 3.4 shows our testbed, which consists of multiple virtual machines (VMs) running on the Amazon cloud. We chose high-end VMs with 1 Gbps links, so that the shared cloud environment does not interfere much with our network setup. The HTTP server in the figure is YouTube, when we actually run the YouTube multiview client. In other experiments, we install and use the nginx \(^1\) as our HTTP server. Users run our multiview player with different rate adaptation algorithms. When we compare against YouTube, we use the player embedded in the Google Chrome web browser. The bandwidth and latency of the network links connecting VMs with the server are controlled using the Linux Traffic Control tc utility. We experiment with multiple network conditions to stress our algorithm.

We use a multiview video released by YouTube [3]. The video is for a concert and it has four different views shot by four cameras. The views cover the singer, band, stage, and fans. The user is allowed to switch among the four views at any time. The video is about 350 sec long, and it has four quality levels \( Q = \{0.5, 1, 1.6, 2.8\} \) Mbps. As of the writing of this thesis, YouTube did not release other multiview videos, and we can not import multiview videos to YouTube because of the proprietary nature of its multiview player.

We consider a general user activity model, which is captured by the Markov chain in Figure 3.2a. The Markov model has four states corresponding to the four views of the video. We control the transition probabilities to analyze the performance under different user activity patterns. Specifically, we create three activity patterns from this Markov chain by adjusting the transition probabilities: (1) FQ, frequent activity, where the user frequently switches views with an average view watching time of 30 seconds, (2) IFQ, infrequent activity, where the number of view switches is much less and the average view watching time is 60 seconds, and (3) GLB, where pattern follows the probabilities in Figure 3.2a. In FQ and IFQ patterns, we assign higher probabilities to switching to next view such that \( p(V_{i+1}|V_i) > p(V_j|V_i), i \neq j \) to make sure that the user will watch the four views during the session. We note that our algorithm is totally unaware of these user patterns initially, but it does try to infer and use them as the session proceeds.

\(^1\)https://www.nginx.com/
We consider multiple performance metrics; most of them were used in similar previous works, e.g., in [39, 91]. In particular, we measure the average of each of the following metrics across all streaming sessions in each experiment:

- **Rendering Quality**: bitrate of the watched views.
- **Rate of Buffering Events**: number of buffering events divided by the session duration.
- **Prefetching Efficiency**: ratio of the used (i.e., rendered) bytes to the total amount of fetched bytes from all views.
- **Fairness (Jain’s) Index**: for $M$ concurrent sessions, we calculate Fairness Index as $JI = (\sum_{i=1}^{M} Q_i)^2 / (M \sum_{i=1}^{M} Q_i^2)$, where $Q_i$ is the average quality in session $i$.
- **Server Load**: total bits sent by the server per second to serve all streaming sessions.

User studies [83, 39] show that the rendering quality and rate of buffering events should be optimized to gain user engagement. Thus, the first two metrics indicate user satisfaction. The third metric shows the bandwidth utilization. The Fairness Index indicates whether competing users get their fair share of bandwidth. The server load represents the bandwidth requirements on the server side to support multiview streaming.

### 3.7.2 MASH vs. YouTube Multiview Rate Adaptation

In this experiment, we compare individual multiview streaming sessions managed by the rate adaptation algorithm of YouTube (denoted by YT), which is proprietary and implemented in the Google Chrome web browser, against the MASH algorithm implemented in our own multiview player. We stream the considered multiview video many times in different network conditions using the two rate adaptation algorithms: MASH and YT.

Specifically, we use the `tc` utility to create five scenarios for the average bandwidth on the link between the client and the Internet, assuming the bottleneck link is the last hop as YouTube is well
E.1.3.5

provisioned: (1) average bandwidth of 8 Mbps, (2) average bandwidth of 16 Mbps, (3) bandwidth changes from 16 to 40 Mbps at $t = 60$ seconds, (4) bandwidth changes from 40 to 16 Mbps at $t = 60$ seconds, and (5) bandwidth changes from 16 to 40 Mbps at $t = 60$ seconds, then from 40 to 16 Mbps at $t = 90$ seconds. In the first two scenarios, the average bandwidth is kept constant, while in the last three the bandwidth changes. Thus, in our experiments, we are capturing the view switching by users as well as the dynamic changes of bandwidth. For each bandwidth scenario and for each user activity pattern, we start the streaming session using the Chrome browser and request the considered multiview video from YouTube. Once the session starts, we open the Developer Tools panel in Chrome to collect HTTP request and response records. During the streaming session, we switch among the views following the user activity patterns. After the session ends, we collect the corresponding HAR file. HAR files store information about the page, HTTP requests and responses, and timing information. We are interested in one object called entries. YouTube client interacts with the server using request query string. Specifically, YouTube client includes 36 query strings in each request for each view. We extract four query strings: (1) id: distinguishes different views, (2) mime: indicates whether this is a video or audio segment, (3) itag: represents the quality profile, and (4) bytesrange: the requested byte range. We focus on video segments only. We also do not pause the video at any time. We repeat the whole experiment using the MASH algorithm. In all following figures, vertical dotted red lines correspond to view switch time.

Figure 3.5 summarizes the results for the FQ user activity pattern, when the client average bandwidth is 8 Mbps. We note that 8 Mbps should be sufficient for YouTube since it naïvely downloads multiple views. If we decrease the bandwidth, YouTube will have lower rendering quality and higher rate of buffering. Figure 3.5a shows that MASH achieves much higher and more consistent rendering quality than YT. The figure indicates an improvement of up to 300% in the rendering quality can be achieved by our rate adaptation algorithm. The maximum improvement is shown in the figure around $t = 210$ seconds. MASH is conservative about fetching segments of different views at high bitrates. YT, however, is aggressive and requests highest bitrates most of the time, even if a view is inactive. This is shown in Figures 3.5b and 3.5c where YT prefetches a lot of bytes most of the time, especially at the beginning. MASH prefetches more bytes only when an activity pattern is captured. For example, Figures 3.5b and 3.5d show that at period [240, 270], MASH prefetches $V_2$ at higher bitrate although the active view is $V_1$. That is why MASH’s rendering quality does not suffer at view switches. On the other hand, Figure 3.5e shows that YT is unaware of such pattern. At the beginning, YT fetches all views at highest bitrates, then it focuses on the current and previous active views only. For example, if the user switches from $V_1$ to $V_2$, YT keeps requesting $V_1$ at high bitrates for a while. This is shown in Figure 3.5e, where YT fetches bytes for inactive views $V_1$, $V_3$ and $V_4$ in period [150, 180] almost as $V_2$. So, when it requests segments for active view $V_2$ in the same period, it takes time to reach a high bitrate. The decisions taken by MASH and YT do not only affect rendering quality, but the rate of buffering and prefetching efficiency as well. YT suffers from playback interruptions as shown in Figure 3.5f, with buffering rate of 0.056 (i.e., 20 re-buffering events in 350 sec), while MASH is perfectly smooth. Similarly, YT prefetches a lot of
bytes without rendering them with prefetching efficiency of 24.3%, whereas for MASH it is 52.7% (Figure 3.5g). For other view switching and bandwidth scenarios, similar prefetching efficiency and rate of buffering results are obtained. YT can achieve the same rendering quality as MASH in high bandwidth scenarios only.

Figure 3.5: Comparison of MASH versus YouTube, for FQ (frequent view switching) scenario and bandwidth = 8 Mbps.
Finally, Figure 3.6 summarizes the average performance across all the experiments. The figures show that MASH does not exhibit playback interruptions while YT’s rate of buffering is up to 0.026, and prefetching efficiency is improved up to 2X.

In summary, our experiments showed that MASH can produce much higher (up to 3X) and smoother quality than YT. They also show that unlike YT, MASH does not suffer from any playback interruptions even in presence of frequent user activities and dynamic bandwidth changes. Moreover, MASH is more efficient in using the network bandwidth, with a prefetching efficiency up to 2X higher than that of YT.

3.7.3 Fairness and Comparisons vs. others

In this experiment, we assess the fairness and scalability of MASH, by analyzing the performance of concurrent multiview streaming sessions. We also compare our algorithm versus others. As shown in Figure 3.4, each of the $M$ VMs concurrently runs our multiview player and requests the video from the server. The upload capacity of the server is $C$ Mbps, and it is shared among all $M$ users. Users have different download capacities. We set these capacities according to the Akamai state of
the Internet report.\textsuperscript{2} Specifically, the probability of $C_i = x$ Mbps follows this distribution $\{0.31, 0.37, 0.13, 0.12, 0.07\}$, where $x \in \{4, 10, 15, 25, 35\}$ Mbps. Similarly, we set the latency using the AT&T global IP network latency averages\textsuperscript{3} with values $\{20, 35, 55, 100\}$ msec following a uniform distribution. For user activities, 60%, 25% and 15% of players follow the GLB, FQ and IFQ patterns, respectively. We use same quality levels $Q$ as the previous experiment. We evaluated MASH with $C = 1$ Gbps and $M = 100$ users.

As elaborated in Section 3.2, we are not aware of any rate adaptation algorithms for multiview video streaming over HTTP in the literature. To partially mitigate this issue, we compare MASH against two variations of current rate adaptation algorithms for single-view videos that can be used, after our modifications, with multiview videos. One might think of these variations as representative samples of possible extensions of current algorithms [73]. Specifically, we consider two buffer-based rate adaptation algorithms: (1) Bl, baseline, that prefetches every view independently whether it is active or not, and (2) Im, inactive minimum, that prefetches the active view based on a buffer-rate function, and all inactive views with the minimum quality $Q_{min}$.


\textsuperscript{3}https://ipnetwork.bgtmo.ip.att.net/pws/global_network_avgs.html
Figure 3.7 shows the cumulative distribution function (CDF) of the server upload bitrate to serve the 100 users by each algorithm. BL requests more data most of the time: more than 500 Mbps 80% of the time and up to 1050 Mbps. Whereas the maximum server upload bitrate is 780 and 470 Mbps for IM and MASH, respectively. The total amount of data during streaming sessions is 253 Gb, 146 Gb and 131 Gb for BL, IM and MASH respectively. Compared to BL and IM, MASH requests less data while improves rendering quality, prefetching efficiency, buffering rate and fairness. Figures 3.8a and 3.8b show that MASH outperforms IM in terms of rendering quality and prefetching efficiency for all streaming sessions. Specifically, MASH improves the average rendering quality and average prefetching ratio by up to 78.5% and 38.6%, respectively. This is because IM fills its inactive buffers with low bitrate segments, hence, the quality drops. Also, when a view switch occurs, most of these segments are not rendered, thus, the prefetching efficiency decreases. Since BL fetches all views all the time, it shows an improvement of the average rendering quality by up to 10%. This, however, comes at the cost of low prefetching efficiency. MASH improves the average prefetching efficiency compared to BL by up to 54.4%. Moreover, MASH improves BL average buffering rate by up to 50% as shown in Figure 3.8c. MASH incurs less buffering events in less number of streaming sessions. In particular, MASH and BL incur 1 and 1.125 buffering events for 6 and 8 streaming sessions, respectively. Since IM prefetches all views, it delivers smooth playback for 99 streaming sessions. However, it fills the buffers of inactive views with low quality segments, hence, it significantly reduces rendering quality compared to MASH (Figure 3.8a). Finally, MASH provides fair share of average quality across concurrent sessions with Fairness Index of 0.93. On the other hand, the index is 0.88 and 0.82 for BL and IM, respectively. This is because these algorithms request segments more than a user may need, hence, resulting in large quality variation across concurrent sessions. While MASH smartly considers user quality needs according to his/her activity pattern, resulting in fair share of high quality for all concurrent sessions.

In summary, our experiments showed that MASH achieves fairness across competing multi-view streaming sessions, and it does not overload the streaming server. Moreover, MASH outperforms other rate adaptation algorithms, which are derived from current adaptation algorithms for single-view videos.

3.8 Summary

Adaptively streaming multiview videos over HTTP is more challenging than streaming single-view videos, because the system needs to support user interactivities as well as handles the network dynamics. Despite the recent interest in the generation of multiview videos, very few works considered designing rate adaptation algorithms for such complex videos. This chapter addressed this problem and presented a novel client-based multiview adaptive streaming over HTTP algorithm (called MASH). MASH achieves high rendering quality, smooth playback and efficient bandwidth utilization by modeling and supporting user interactivities. Specifically, MASH constructs and combines local and global view switching Markov chains to weigh the importance of views in the video.
We showed that these models converge and impose low overheads on the client and server. We presented a new buffer-based approach to request segments based on the relative importance of different views and the current network conditions. We implemented the proposed algorithm and compared it against the rate adaptation algorithm used by YouTube to stream multiview videos. Our extensive empirical evaluation showed that MASH substantially outperforms the YouTube algorithm in terms of rendered quality, number of buffering events, and prefetching efficiency. In addition, our results showed that MASH: (i) is scalable as it does not overload the server, (ii) achieves fairness across concurrent streaming sessions, and (iii) renders smooth and high quality even in presence frequent view changes.
Chapter 4

Joint Content Distribution and Traffic Engineering of Adaptive Videos in Telco-CDNs

In this chapter, we propose a new algorithm to address the problem of joint traffic engineering and content distribution of adaptive videos in telco-CDNs. We then describe the implementation of the proposed algorithm in a realistic testbed, and compare it against the closest work in the literature.

4.1 Introduction

The amount of multimedia traffic distributed over the Internet has been increasing at high rates in the past several years [1]. Content Delivery Networks (CDNs) have played a critical role in supporting this increasing demand. Current CDNs, e.g., [107], replicate content at different caching locations and direct users to the closet/best location based on various factors such as geographical distance, end-to-end latency, and traffic load at different locations.

To improve user-perceived quality, current CDNs often infer network conditions inside ISPs through complex measurement methods [107, 50]. However, CDNs cannot control the traffic flows and what paths they take inside ISP networks, which ultimately carry the actual traffic. CDNs may not even precisely know the underlying network topology and the current traffic situation on links and switches in the ISP network.\(^1\) Thus, decisions made by CDNs may negatively impact the load on various links in the ISP networks [94], especially links between different network domains (called inter-ISP links), which are costly [105]. This may trigger ISPs to adjust traffic routing inside their networks, which in turn, could impact the performance of CDNs. To reduce the mis-match between the goals of CDNs and ISPs, multiple ISP-CDN collaboration models have been proposed, from defining interfaces that ISP and CDN can use to share information [49], to allowing content providers to deploy caches within ISPs (e.g., Netflix OpenConnect [103]). While useful, these ap-\(^1\) Although there are tools for inferring ISP topologies [122] and estimating network traffic [96, 117], they are, in general, quite expensive and cannot be used for obtaining accurate information about large-scale ISPs in real-time.
proaches can only provide partial solutions. For example, OpenConnect caches [103] can provide local access to popular content within the ISP, but they cannot control the network paths taken to carry the traffic.

The difficulties of enabling ISP-CDN collaboration and the potential business opportunities motivated major ISPs, e.g., AT&T, to deploy and manage CDNs inside their networks [11, 29]. Such CDNs are referred to as telco-CDNs. Telco-CDNs offer unprecedented opportunities for optimizing the delivery of multimedia content, because not only can they jointly leverage the up-to-date information about the underlying ISP network and the characteristics of the multimedia objects, but they can also choose the appropriate paths for different sessions and configure the network to enforce these decisions. Managing telco-CDNs is, however, a complex task, because it requires concurrently managing resources at the network layer (traffic engineering (TE)) and system layer (processing and storage capacities of caches), while supporting the adaptive nature and skewed popularity of multimedia content.

In this chapter, we propose a comprehensive solution to efficiently solve the resource management problem in the emerging telco-CDNs, which we refer to as CAD (short for Cooperative Active Distribution). CAD has two goals: (i) reducing the cost incurred by ISPs, and (ii) improving the quality of multimedia sessions by reducing end-to-end latency. CAD achieves these goals by serving as much as possible of the multimedia traffic locally within the ISP, while carefully engineering the traffic paths to avoid creating bottlenecks in the network. CAD can significantly improve the performance of multimedia streaming systems in scenarios where ISPs and content providers cooperate (e.g., Netflix OpenConnect [103]), and when ISPs own their streaming services such as AT&T U-Verse and Bell Canada. Both scenarios are increasingly seeing wider adoption and deployment in practice.

We present multiple novel ideas in CAD, and we believe that each of them can be useful in its own right. For example, we consider fine-grained caching of multimedia objects. In particular, in current streaming systems, e.g., Netflix, a multimedia object is transcoded into multiple forms of qualities/resolutions/formats (referred to as representations) in order to support a wide-range of users with heterogeneous devices and dynamic network conditions. Instead of considering the popularity at the level of multimedia objects, we account for the popularity of individual representations of each object. This is important as some representations are requested much less than others, e.g., a representation customized for uncommon display or rare network condition. In addition, major ISPs deploy storage infrastructures and general-purpose multi-core servers inside their networks [50]. We leverage this processing power in creating some representations on demand, instead of pre-creating and storing all of them. This can save inter-ISP bandwidth, as a requested representation that is not locally stored can be created by transcoding another representation instead of having to fetch it from outside the ISP. This, however, needs to be done carefully so that the additional delay introduced by the on-demand creation of representations does not exceed a predefined threshold. Another novel aspect of our solution is that it enables the cooperation among caches within the ISP not only to
serve representations, but also to create them on demand. Finally, we consider the active creation and cooperative serving of representations in the computation of the traffic paths in the network.

We have implemented CAD and evaluated it on top of Mininet [61], which processes and carries real traffic. We implemented caching servers that stream multimedia content using the widely-used DASH protocol [123]. We used virtual switches to realize the TE aspects of CAD, which are enforced using SDN (Software-defined Networking) rules. We implemented an OpenFlow controller to manage the whole telco-CDN. We used an actual ISP topology and conducted extensive experiments to assess the performance of CAD and compare it against the closest work in the literature [119]. Our results show substantial improvements across multiple performance metrics. For example, CAD achieves up to 64% reduction in the inter-domain traffic and 14% improvement in the buffer occupancy (i.e., number of buffered segments) at clients compared to the algorithm in [119]. Buffer occupancy indicates a better quality of experience (QoE) for users in adaptive streaming [73]. Our results also show that CAD does not overload the intra-domain links. This ensures the stability of the inter-domain routing policies employed by the ISP [127].

The rest of this chapter is organized as follows. We summarize the related works in Section 4.2. Then, we present the system models and problem definition in Section 4.3. We describe the details of the proposed algorithm CAD in Section 4.5. We present our implementation and evaluation results in Section 4.6, and we conclude the chapter in Section 4.7.

4.2 Related Work

There has been a large body of work on ISP collaboration opportunities, content distribution, TE, telco-CDNs and online transcoding systems. We summarize the relevant works to CAD in the following.

**ISP–CDN Collaboration.** Most current CDNs are run independently from ISPs that own and operate the underlying networks. Thus, such CDNs may not have access to accurate and current information about the network topology and traffic conditions [107, 49]. This may result in higher costs and lower performance for both the CDN and ISP. To overcome these problems, works such as NetPaaS [49] enable ISP-CDN collaboration. These works focus on *information sharing*. In contrast, CAD *jointly* solves the content distribution and TE problems, which is a more challenging problem due to the heterogeneity of content and its popularity.

Major content providers impose significant load on ISP networks. For example, IX Australia reported that the peering traffic increased by 100% for some ISPs in Australia because of the high demand of streaming Netflix videos.² To partially mitigate this high load, Netflix started installing caches inside ISPs and deploying peering links to Internet eXchange Points (IXPs). Similarly, the Streaming Video Alliance has recently started trials to bring caching servers to ISP networks [124].

²http://bit.ly/2htgP4b
Unlike CAD, most of these caches work independently, and they cannot control the network paths taken by streaming sessions within the ISP.

**Joint Caching and TE.** Xie et al. [141] formulated the TE-guided collaborative caching (TECC) problem for in-network caching systems. TECC only calculates the amount of traffic for every source-destination pair along a direct path. CAD, in contrast, supports multipath routing, and calculates network paths for every source-destination-video tuple. CAD utilizes the processing power at caching sites and the relationship between representations to create some of them on demand. Jiang et al. [79] developed a collaboration mechanism between ISPs and content providers to coordinate the TE and server selection, assuming that the content is already placed at caching servers.

**Telco-CDNs.** Telco-CDNs share the vision of the recent proposal in [9], which allows content providers to reach inside ISPs and control their own traffic flows through network virtualization to optimize the user experience. The work in [9] only outlines the vision and its potential, but it does not provide algorithmic solutions to manage caching resources and network traffic. Li and Simon [90] proposed a push-based approach to manage telco-CDNs, but it does not manage the network resources. Zhou et al. [152] proposed an algorithm to maximize the number of live streaming sessions served with the lowest bandwidth in telco-CDNs. In contrast, CAD manages telco-CDNs to support on-demand streaming.

**Online Transcoding Systems.** Cloud transcoding systems have emerged to create video representations on demand. Gao et al. [52] jointly scheduled transcoding tasks and provisioned computation resources to maximize revenue and satisfy user requirements. Krishnappa et al. [84] proposed an online transcoding system for CDNs. Ma et al. [95] built a transcoding time estimator and integrated it with task scheduler to reduce transcoding completion time. These systems do not consider cooperation between servers nor do they support TE.

**In summary,** unlike previous systems from industry and academia, CAD simultaneously considers: (1) the joint optimization of caching resources and the engineering of traffic paths in the network to serve adaptive multimedia content, (2) the available processing capacity at caches to create video representations on-demand, and (3) the cooperation among caches to serve and create video representations on demand.

### 4.3 System Models and Problem Definition

In this section, we describe the considered multimedia content model and the architecture of the telco-CDN. Then, we state the problem addressed in this chapter.

**4.3.1 Multimedia Content Model**

We consider multimedia objects served using adaptive streaming protocols, such as DASH [123], to support heterogeneous clients and accommodate dynamic network conditions. In this model, a multimedia object (e.g., video) is encoded in multiple *representations* based on several factors, including available bandwidth, screen resolution, screen type, and media player. Some representations
require transcoding the video into lower bit rate, which is computationally expensive, while others may require simpler operations such as changing the file format. All representations are created from the original copy, which is referred to as the master representation. Each representation is divided into equal-length segments in the order of one to a few seconds. In most current CDNs, all representations are pre-created and stored before starting the distribution process. In the proposed telco-CDN, we create some of the representations on demand inside the ISP, which can be done given the target environment for CAD is telco-CDN where the content and the network resources are managed/owned by the same entity (or cooperating entities).

Representations of the same multimedia objects may have different popularity. For example, a representation customized for uncommon network conditions will see fewer client requests compared to a representation customized for typical network conditions. CAD considers the popularity of individual representations, which provides more room for optimizations compared to the current model of considering the popularity of a multimedia object without accounting for the relative importance of its various representations. The popularity of a representation is defined in terms of the number of requests for that representation during the previous periods.

In the current and widely deployed DASH protocol, each client dynamically chooses the most suitable representation based on the receiving device characteristics and network conditions. CAD does not change the client adaptation method, nor does it require any modifications to the DASH protocol. Rather, CAD efficiently manages the telco-CDN such that clients obtain the requested objects with shorter latency from local caches through carefully-engineered network paths. This may indirectly enable the client adaptation method to request better quality representations and improve the QoE for users.
4.3.2 Telco-CDN Model

A simplified view of the considered telco-CDN is illustrated in Figure 4.1, where an ISP deploys caching servers at different sites to serve multimedia objects to clients with short latency, while minimizing the traffic load on inter-ISP links. The telco-CDN is managed by the proposed CAD (Cooperative Active Distribution) algorithm, which runs on a server within the ISP. The multimedia objects are provided by content providers interested in the delivery services offered by the telco-CDN. Content providers encode each multimedia object in multiple representations. The expected demand for each representation is estimated using predictive techniques such as [51, 32], and can be performed either by the telco-CDN or the content provider. We do not address demand estimation methods in this chapter; our proposed CAD algorithm can work with any demand estimation method. We design CAD to augment streaming systems where the demand of multimedia objects can be estimated. For example, Netflix and Hulu may estimate content demands based on its release cycles. Note that these systems are increasingly attracting viewership, and they have agreements with major ISPs. Extending CAD to work in user-generated video streaming systems is left as a future work.

The design of CAD is motivated by the deployment of microdatacenters by major ISPs [50], where some servers in these microdatacenters can act as caches. Caches in CAD are active and cooperative. They are active because they can create video representations using their own processing capacity, provided that the time to create these representations does not exceed a predefined threshold latency $L$ seconds. The cooperative aspect of CAD allows one cache to fetch a requested representation from another, provided that appropriate network paths can be found to support the requested session. If a representation cannot be served from caches within the ISP, it will be requested from the content provider. This is all done transparently to end users.

The active and cooperative features of the proposed telco-CDN allow it to serve as much of the client requests as possible within the ISP, which reduces the load on inter-ISP links and shortens the latency. In addition, inter-ISP links are usually the main points of congestion in the Internet where packet losses and long queuing delays happen [105]. Thus, by reducing the amount of traffic across inter-ISP links, the proposed telco-CDN reduces the chances of packet losses and further shortens user-perceived delays, which improves the QoE for users.

4.3.3 Problem Definition

We consider a multi-region ISP operating a telco-CDN, which is modeled as a graph $(\mathcal{S}, \mathcal{E})$, where $\mathcal{S}$ represents caching sites and $\mathcal{E}$ represents links between the sites. Each caching site $s \in \mathcal{S}$ has one or more servers with aggregate storage capacity $b[s]$ bits and processing capacity $p[s]$ cycles/sec. Each link $e \in \mathcal{E}$ has a capacity of $c[e]$ bits/sec. The set of multimedia objects (videos) is denoted by $\mathcal{V}$, where each video $v \in \mathcal{V}$ has up to $R$ representations. The popularity (or demand) of a video representation $r$ at caching site $s$ is denoted by $d[s][r]$ bits/sec.
The problem we address in this chapter is to serve user requests from the telco-CDN by allocating the storage and processing resources at each caching site to the different video representations as well as directing the traffic flows of the video sessions through the ISP network such that the cost incurred by the telco-CDN is minimized (in terms of the amount of inter-ISP traffic) and the user-perceived quality is optimized (in terms of end-to-end latency). We divide this problem into two sub-problems: creating distribution plan (referred to as DP) and deciding the network paths for the video sessions (referred to as TE). The DP sub-problem determines for each caching site which video representations to store locally so that they can readily be served to clients near to that site, and for the remaining video representations which ones to: (i) create on demand, (ii) fetch from another caching site within the ISP, or (iii) fetch from the origin server. The TE sub-problem computes network paths to minimize the maximum link utilization (MLU) inside the ISP network. With few modifications to the proposed solution, other TE objectives can be plugged into the proposed system.

4.4 Overview of CAD

The proposed CAD algorithm jointly solves the DP and TE sub-problems. At high level, CAD runs periodically on a server inside the ISP. The period is a configurable parameter and can range from tens of minutes to several hours. In each period, CAD uses information about the popularity of various video representations and available resources inside the ISP (CPU cycles, storage, and available bandwidth on different links) to compute the distribution plan for all representations. The distribution plan specifies for each caching site which representation to store, create on demand, fetch from other caching sites, or fetch from the origin server. During the computation of the distribution plan, CAD considers the available bandwidth in the calculation and it specifies the network paths within the ISP to be taken to serve each video session, that is, it also solves the TE sub-problem while solving the DP sub-problem.

CAD produces two outputs: distribution plan and traffic engineering rules. The distribution plan is sent to the caching sites so that each server knows what to do upon receiving a user request. The traffic engineering rules are SDN rules that are sent to routers in the ISP network so that they can enforce the decisions made by CAD. The actual video sessions are created only when users submit requests. When a user submits a request for a representation, the request is directed to the caching site nearest to the user, using methods commonly used in current CDNs such as DNS re-direction. Using the pre-computed distribution plan, the caching site can handle the user request locally, from other caches, or from origin server.

Jointly computing the distribution plan and TE rules of CAD is challenging. Specifically, representations are heterogeneous in terms of popularity, and storage and processing requirements. For example, creating a popular representation on-demand may result in service delays. In addition, cooperative and active distribution results in traffic inside the ISP network that needs to be routed efficiently. Moreover, single path routing may result in congestion at links inside the network. Fi-
nally, CAD should compute the distribution plan and TE rules in a timely manner (i.e., within the period configured by the telco-CDN operator).

We introduce multiple ideas to address these challenges. CAD computes for each representation a value that captures its importance based on its popularity, storage and processing requirements. Given the demand for a representation, we create a weighted sub-network that spans the destination (i.e., demand site) and the sources (i.e., sites where the representation or its master copy are stored). Then, we transform the TE problem to a min-cost flow problem in this sub-network. This allows CAD to support multi-path routing based on link usage to balance the total traffic across all links, and reduces the running time of the proposed algorithm.

4.5 Details of CAD

A high-level pseudo code of the proposed CAD algorithm is shown in Algorithm 2. CAD computes and returns two outputs: \(D\) and \(T\). \(D\) is a data structure with a record \(D[s]\) for each caching site \(s \in S\). \(D[s]\) has one entry for each video representation \(r\), which contains two fields: \textit{status} and \textit{sites}. The \textit{status} field indicates whether the caching site \(s\) should store, create on demand, fetch from other caching sites, or fetch from the origin server the video representation \(r\) when a client submits a request for that representation. The \textit{sites} field contains a list of caching sites to obtain a representation or its master copy from in case the \textit{status} of that representation is set to fetch from other sites. The second output of CAD is \(T\), which is a list of traffic engineering rules for routers in the ISP network. Each rule is in the form of \(\langle RID, src, dst, iport, oports, weights \rangle\), where \(RID\) indicates the ID of the router to execute this rule. A TE rule specifies that the traffic flow coming from the \(src\) address on input port \(iport\) and going to the \(dst\) address should be forwarded on the output ports \(oports\) with different ratios specified by the \textit{weights}. In the initialization stage, the status of all representations is set to fetch from the origin server, and the TE rules are set to the default ones used by the ISP (e.g., shortest paths computed by the intra-domain routing protocol).

CAD maintains multiple data structures, including 2-dimensional matrix \(V\), list \(\text{sites}[r]\) for each representation \(r\), and the aggregate traffic map \(F\). For every caching site \(s \in S\), \(V[s]\) is a list that sorts all video representations descendingly based on their values. The value of a representation \(r\) at caching site \(s\) is given by \(d[s][r] \times \text{cpu}(r)/\text{size}(r)\). This value determines the priority of storing the representation at a caching site such that a popular small-size representation with high processing requirement has higher priority to be stored. The list \(\text{sites}[r]\) maintains the caching sites that \(r\) is stored at. The aggregate traffic map \(F\) contains the traffic value for every src-dst pair. Specifically, if \(F[src, dst][i, j] = f_{(i,j)}\), then the aggregate traffic from router \(i\) to router \(j\) for demands from \(src\) to \(dst\) is \(f_{(i,j)}\) units of traffic.

As shown in Algorithm 2, CAD starts with an initialization stage where all representations are sorted based on their values and the result is stored in \(V[s]\). In addition, for every site \(s \in S\), CAD sets \(D[s]\) to store as many representations as the storage capacity \(b[s]\) allows, and keeps an index of
Algorithm 2 CAD: Calculate $D$ and $T$

Require: $T$: ISP network topology
Require: $d[s][r]$: demand for representation $r$ at caching site $s$
Ensure: $D$: distribution plan
Ensure: $T$: traffic engineering rules

Initialization:
1. Calculate rep. value per site, and add it to $V[s]$
2. Sort all representations in $V[s]$ by their values, $\forall s \in S$
3. Iterate over all sites $s \in S$ and representations in $V[s]$
   
   If $\text{size}(V[s][i]) \leq b[s]$
   
   $D[s][i].\text{status} = \text{stored}$,
   
   sites[$r$].append($s$),
   
   $b[s] = \text{size}(r)$; last_stored($s$) = $i$
4. $\gamma_{\text{current}} = 0$ // Current MLU is set to zero
5. $F = \{\}$ // Aggregate traffic map

1: for ($s \in S$) do
2:   for ($i = \text{last} \_\text{stored}(s)+1$ ; $i < \text{rep} \_\text{count}$; $i++$) do
3:   $r = V[s][i]$
4:     Construct sub-network $T'$
5:     $f = \text{FINDPATHS}(s, d[s][r], T', \text{sites}[r])$
6:     $c = \text{CREATEREPAT}(s, r)$
7:     decision = $\min(f, c)$ // either $f$ or $c$ based on MLU
8:     if (decision) then
9:         $\gamma_{\text{current}} = \text{decision,}\gamma$
10:        $\text{UPDATEDPANDTE}(r, \text{decision,} F)$
11:     end if
12:   end for
13: end for
14: $T = \text{CALCTERULES}(F)$
15: return $D$ and $T$
the last stored representation in last_stored(s) variable. CAD also updates the list sites[r] for each stored representation r. The current MLU γ_{current} is set to zero.

**Calculating the distribution plan.** After initialization, CAD attempts to update D for every representation at every site to either fetch or create and calculate T accordingly in Lines 1 to 13. For every site s, the algorithm iterates over all representations starting from the index last_stored(s)+1. In every iteration, the algorithm calls two functions: FINDPATHS and CREATEREPAT. Each function returns caching sites, network paths and the resulting maximum link utilization γ. A returned γ value of ∞ means no solution was found. The FINDPATHS function requires creating a sub-network T' (Line 4). It attempts to find TE paths that accommodate the demand of r from the caching sites stored in sites[r] to the local site s. The CREATEREPAT function checks if r can be created locally. After the two functions return, CAD chooses the option that minimizes the maximum link utilization γ in Line 7. CAD then updates the current MLU value γ_{current}, and calls UPDATEDPANDTE to update D and F.

First, the function UPDATEDPANDTE updates the corresponding D[s] status to either fetched or created based on the decision variable, and sets the caching sites. Second, based on the returned network paths, it aggregates the traffic between every two routers per src-dst pair in the F map. It also decreases the available bandwidth for every link. Third, if r is to be created on-demand at caching site s, it decreases the available processing capacity p[s].

We provide more details on the FINDPATHS and CREATEREPAT functions. The goal of FINDPATHS is to balance the traffic across all links in the network given a demand value d[s][r] and stored representations at sites[r]. To achieve this TE goal, we transform the TE problem to the min-cost flow problem in the sub-network T' as follows. We first construct a sub-network T' = (S', E') with vertexes and edges contained in all paths from sites[r] to s. Only edges (links) with remaining capacity greater than zero are included. The cost of every edge e in the sub-network is calculated as \( \omega'_e = d[s][r]/c[e] \). Then, the costs of inter-ISP links are increased to double of their value in order to reduce the amount of inter-ISP traffic, because paths are selected based on lower costs. Finally, we solve the min-cost flow problem (MCF) using the cost-scaling algorithm in [53]. The algorithm returns a set of sites that will transmit the representations, a set of links with their traffic values, and the resulting maximum link utilization γ. To calculate γ, we iterate over all links in the output set of links, divide the total traffic for every link by its capacity, and choose the maximum value. We note that the cost-scaling algorithm requires integer edge costs. To meet this requirement, we multiply all costs by a relatively large integer (e.g., 100) and round up all costs to the nearest integer values.

The function CREATEREPAT is described in the pseudo code in Algorithm 3. It checks if a representation r can be created on-demand locally within the allowed delay L seconds. First, it calculates the required CPU cycles to process d[s][r]. To calculate the cycles, it computes the segments count of r to be requested per second at caching site s. Then, it multiplies the processing requirement per segment by the segments count. If the required cycles are less than the available CPU resources in L seconds (i.e., p[s] × L), the algorithm checks if the master copy is stored locally, and returns the site s as the a caching site. Since the master copy is stored locally, the MLU is not affected and we
Algorithm 3 Find sites and flows for creating $r$ at $s$

1: function \textsc{CreateRepAt}(s, r)
2: \hspace{1em} seg\_count = d[s][r]/\text{seg\_size}(r)
3: \hspace{1em} cycles = cpu(r) \times \text{seg\_count}
4: \hspace{1em} if (cycles $\leq$ $p[s] \times L$) then
5: \hspace{2em} if ($s \in \text{sites}[r\text{-master}]$) then
6: \hspace{3em} return sites=$\{s\}$ and links=$\{}$ and $\gamma_{\text{current}}$
7: \hspace{2em} else
8: \hspace{3em} $m = d[s][r] \times \text{seg\_size}(r\text{-master})/\text{seg\_size}(r)$
9: \hspace{3em} Construct sub-network $T'$
10: \hspace{3em} $f = \text{FindPaths}(s, m, T', \text{sites}[r\text{-master}])$
11: \hspace{3em} if $f.\gamma < \infty$ then
12: \hspace{4em} return $f.\text{sites}$ and $f.\text{links}$ and $f.\gamma$
13: \hspace{3em} end if
14: \hspace{2em} end if
15: \hspace{1em} end if
16: \hspace{1em} return sites=$\{}$ and links=$\{}$ and $\gamma = \infty$
17: end function

return the current MLU. If the master copy is not stored locally, the algorithm checks if the network can accommodate fetching it from sites[r.master]. It then calls \textsc{FindPaths} to find TE paths. If the network can fetch r.master, then the corresponding sites, links and MLU are returned.

**Calculating the TE rules.** CAD aggregates the traffic between every two routers per src-dst pair using the aggregate traffic map $F$. Then it calls \textsc{CalcTERules} in Line 14 to find the TE rules. \textsc{CalcTERules} takes $F$ as an input and calculates the output port weights for every router per src-dst pair in the TE paths. To do so, \textsc{CalcTERules} iterates over all src-dst pairs in $F$. For a router $i$, it finds all $F$ entries where router $i$ has output flows. To find the weight of port $n$ that connects $i$ to $j$, we divide the $i,j$ output flow value over the sum of all output flow values for router $i$. That is $T[src, dst][i][n] = F[src, dst][i,j]/\sum_k F[src, dst][i,k]$.

The computed TE rules by CAD is abstract enough to be used by various TE technologies such as OpenFlow [98], or Segment Routing [46]. In OpenFlow-managed ISPs, the TE rules can be installed as follows. The matching rules are source and destination, and the action is to process the packet using a Group Table. The type of the Group Table is \texttt{select} that allows weighted round robin processing of the packets using multiple buckets in the group. The bucket count is equal to the port count that have weights greater than zero in the TE plan. Each bucket $n$ is associated with an Output action to port $n$ with weight $T[src, dst][i][n]$. Segment Routing, on the other hand, does not require additional state at routers. It attaches to each packet a set of labels that represents the source-destination path.

**Time Complexity.** In the following lemma, we show that CAD terminates in polynomial time.

**Lemma 2.** CAD terminates in polynomial time in the order of $O(N \log(D))$, where $N$ is the total number of representations and $D$ is the maximum demand.
Proof. There are three main functions in CAD. We first show that they terminate. Then we compute the worst-case running time. Let $K$ (a constant) be the number of caching sites. The first function is \textsc{FindPaths} that computes TE paths using the cost scaling algorithm. Since we construct the sub-networks such that their link capacities and costs are integers, the cost scaling algorithm is guaranteed to terminate \cite{53}. The function \textsc{CreateRep} terminates because it only calls \textsc{FindPaths} and performs few other steps. Finally, in Algorithm 2, for each caching site, CAD traverses at most $N$ representations. Thus, the total iterations count is bounded by $KN$. Hence, the solution terminates.

The worst-case running time happens when CAD calls \textsc{FindPaths} to solve the min-cost flow problem every iteration. The cost-scaling algorithm finishes in $O(n^2 m \log(nT))$ \cite{53}, where, for our problem, $n$ and $m$ are the caching sites and link counts in the sub-network, and $T$ is the maximum link cost. In CAD, $n \leq K$ and $m \leq n^2 \leq K^2$. The maximum cost is $D = \max_{s \in S, r \in r} d[s][r]$. Thus, the running time is $O(NK n^2 m \log(nD)) = O(NK^3 \log(KD))$. For a telco-CDN, the number of caching sites $K$ does not change frequently, hence, the running time is $O(N \log(D))$. \hfill \Box

We note that the running time of CAD depends on the cost-scaling algorithm to find TE paths. As a result, CAD is a weakly polynomial time algorithm since its running time is polynomial in $\log(D)$. This should not be confused with pseudo-polynomial algorithms whose running times are in polynomial time of the input integer values. On the practical side, $K$ is typically a small constant value compared to $N$ (from a handful of caching sites to up to few tens of sites in large telco-CDNs), and $N$ is in the order of thousands of representations of different videos. Thus, CAD is a practical algorithm that can run on commodity servers.

4.6 Evaluation

In this section we describe the implementation of the proposed CAD algorithm in a realistic testbed on top of Mininet. Then, we compare CAD against the closest work in the literature (referred to as rCDN). Our results show that CAD outperforms rCDN in terms of inter-domain traffic and buffer occupancy at clients by up to 64% and 14%, respectively. In addition, we demonstrate that CAD strictly respects the maximum allowed processing latency set by the operator, and we assess the practicality of CAD in terms of its running time.

4.6.1 Implementation

We implemented all components of the proposed CAD algorithm, from computing distribution plans and traffic engineering rules, to executing the distribution plans by caching sites and enforcing the rules by the ISP routers in real time. Our implementation handles real multimedia traffic, where clients stream content using the DASH protocol. To rigorously evaluate the performance of CAD in different settings, we implemented a configurable framework on top of Mininet \cite{61}. Unlike network simulators, Mininet is a realistic network emulation environment built on Linux namespaces.
Applications running on Mininet can be deployed on actual systems. Mininet has been used to evaluate research ideas at the network level e.g., [96], and application level e.g., [16].

Our Mininet framework emulates an ISP network of a telco-CDN, where the network topology is given as input to the framework. An ISP network is composed of points of presence (PoPs) at different geographical locations, where clients can access the Internet. We abstract the internal details of a PoP and implement it as a virtual switch using Open vSwitch 2.5.1 [110]. Links between virtual switches are created based on the input topology. The latency on a link between two switches is calculated based on the link medium and length. Switches and links process and carry real traffic using the Linux networking stack. An example ISP topology is shown in Figure 4.2, which is the AT&T topology in the East Coast of the US.\(^3\) We use this topology in our experiments, because AT&T manages its own CDN [11]; hence, our setup is close to a real telco-CDN. We note that ISPs inter-connect with each other, typically at IXPs. Our framework emulates these inter-ISP links and measures the load on them, because this inter-ISP load represents an important cost factor for ISPs. For the example topology in Figure 4.2, the three PoPs at which AT&T inter-connects with other ISPs are marked by larger blue icons. The inter-connections occur through three IXPs: Telx Internet Exchange in Atlanta, Equinix Exchange in Chicago, and CoreSite Any2 Exchange in Boston.

The caching sites of the telco-CDN are located at PoPs. We implement each caching site as an HTTP server that can stream multimedia content using the DASH protocol. The characteristics of each caching site (e.g., storage and processing capacities) are provided as input to our framework. Caching sites receive and execute the distribution plans computed by CAD. The CAD algorithm itself is implemented and executed on a separate server inside the ISP. We implemented the streaming DASH clients using Go 1.7.4. The DASH clients request video representations from nearby caching sites according to the input demands. Finally, we manage the ISP network using OpenFlow [98]. In particular, we implemented an SDN controller in Python using Ryu 4.9. The SDN controller installs the TE rules computed by CAD into the switches.

### 4.6.2 Experimental Setup

We compare CAD against the algorithm in [119]. This algorithm reactively serves users using shortest path inside the network. We compare against this algorithm because it was shown in [119] that it outperforms other approaches. We refer to this algorithm as reactive CDN (rCDN). Both CAD and rCDN algorithms are used to manage the telco-CDN of the ISP topology in Figure 4.2, which has 16 locations.

We consider three important performance metrics that measure the network load and user QoE:

- **Amount of Inter-domain Traffic**: the total bytes fetched from the origin server through inter-ISP links.

\(^3\)http://www.topology-zoo.org/dataset.html
Figure 4.2: The ISP topology used in our experiments. It is the AT&T network in the East Coast of the US.

- **Maximum Link Utilization (MLU):** the 95-percentile of the traffic per second passing through an intra-domain link divided by its capacity. We use the 95-percentile because it is a good representation of the overall link performance [67].

- **Buffer Occupancy:** the average number of video segments in the playback buffer at clients. High buffer occupancy values indicate better QoE for two reasons. First, because CAD does not change the client adaptation algorithm, high buffer occupancy may indirectly allow the client adaptation algorithm to request better quality representations [73]. Second, high buffer occupancy reduces the chance of rebuffering at clients.

There are 1,000 videos each with up to 7 representations in the telco-CDN with a total size of 6 TB (i.e., the telco-CDN manages 7,000 representations). These representations offer different quality levels, and they have average bitrates of 5, 3.5, 1.5, 0.5, 0.3, 0.2 and 0.1 Mbps. We transcode the master representation (5 Mbps) to create other representations using FFmpeg, and we measure the average processing time and file size for each representation. We limit FFmpeg processing to one CPU thread, and run each transcoding multiple times and take the average. According to DASH, each representation is divided into equal-size segments, 1 second each. We set the maximum allowed processing latency to 5 seconds.

The video representations in the telco-CDN are requested by many users. We emulate a realistic behavior of users based on the analysis in [12]. Specifically, we consider four characteristics when generating user requests: (1) peak weekly demands, (2) peak daily demands, (3) user behavior of quitting streaming sessions, and (4) representation popularity distribution. We generate user
requests for three days: Friday, Saturday and Sunday, because they represent the peak weekly demands [12]. The request rate is not constant during the day, but it increases until its peak value in the evening. We employ the mixture probabilistic model that was observed in [12]. This model, illustrated in Figure 4.3, consists of three components representing early-quitters, drop-out users and steady viewers. Finally, the representation popularity is assumed to follow a Zipf-Mandelbrot distribution [63], which is a generalized form of the Zipf distribution. Unlike the Zipf distribution which is characterized by a single skew parameter $\alpha$, the Zipf-Mandelbrot distribution has two parameters: skew parameter $\alpha$ and plateau parameter $q$, which captures the fact that multiple objects can have close popularity, which is more realistic. Large $q$ values indicate that the total requests are spread on many videos, while small $q$ values mean that most of the requests are concentrated in few, very popular, videos. When $q = 0$, the Zipf-Mandelbrot distribution reduces to Zipf distribution.

Putting all pieces together, we first generate user requests for each day using three Poisson processes with arrival rates $\lambda_1$ during early hours of the day, $\lambda_2$ during mid-day hours, and $\lambda_3$ during evening hours. Table 4.1 shows the $\lambda$ values used in our experiments per second per region. When a user arrives, we randomly pick a representation following the Zipf-Mandelbrot distribution, and the number of segments the user will watch during the session using the mixture probabilistic distribution discussed above.

The key parameters we control and vary in our experiments are the storage capacity at each caching site and the popularity of video representations. The storage capacity ranges from 1% to 10% of the video library size. For the representation popularity, we fix the skew parameter of the Zipf-Mandelbrot distribution to $\alpha = 1.5$, but we change the plateau parameter $q$ from 1 to 10. All
other parameters in our setup are fixed as follows. Each caching site has 16 CPU cores with clock speed of 2.6 GHz. The capacity of inter-domain links is 400 Mbps and the average delay on them is 50 msec. The capacity of intra-domain links is 500 Mbps, and the delay is proportional to site distances. We repeat each experiment five times with different random seeds, and we report the average of the performance metrics across them.

The telco-CDN goes through two phases every day during our experiments. The first one is the warm-up phase, where caches are populated with videos either using CAD or rCDN. The second phase is the streaming phase, where clients request video segments and caching sites respond with the corresponding content according to the pre-computed distribution plans. We report the performance metrics during the streaming phase.

### 4.6.3 Results

**CAD Outperforms the Closet Algorithm in Literature.** We start by comparing the performance of CAD versus rCDN under different storage capacities per caching site. We set the plateau parameter of the video representation popularity distribution $q = 5$. For legibility of the figures, we only plot the results for the storage capacities of 2%, 4%, 6%, 8% and 10% of the video library size.
Similar results were achieved for the other values of the storage capacity. The average results are shown in Figure 4.4. In terms of the amount of inter-domain traffic, CAD outperforms rCDN by up to 30% when the storage capacity is 10% of the video library size as shown in Figure 4.4a. In addition, CAD improves the buffer occupancy at clients by up to 12.5% compared to rCDN because CAD reduces end-to-end latency by keeping many requests inside the ISP network as shown in Figure 4.4b. Moreover, CAD does not overload the intra-domain ISP links as shown in Figure 4.4c, even when fetching parent or leaf representations from inside the network. In particular, CAD either improves the MLU or results in the same MLU as rCDN. This is because CAD jointly manages the network and caching resources.

A more focused sample of our results is depicted in Figure 4.5, where we show how the amount of inter-domain traffic and buffer occupancy change throughout the 3-day period of the experiment. The results in Figure 4.5 are for $q = 5$ and storage capacity of 10% of the video library size. Similar results were obtained for other values of the storage capacity and plateau parameter $q$. Figure 4.5a depicts the time series of the inter-domain traffic. Vertical arrows show the maximum improvements achieved by CAD for every day. On Friday, both CAD and rCDN result in low inter-domain traffic, because most demands can be handled inside the network. CAD, however, results in much lower inter-domain traffic compared to rCDN. On Saturday, the inter-domain traffic increases for both rCDN and CAD till early hours of the evening, because of the increase in the demands. However, CAD outperforms rCDN in terms of inter-domain traffic and buffer occupancy as shown in Figures 4.5a to 4.5b. In the evening, rCDN results in more requests to the origin server and increases the end-to-end latency. As a result, CAD improves the buffer occupancy compared to rCDN. The same results are observed on Sunday. Specifically, CAD reduces the inter-domain traffic by up to 83%, 57% and 46% on Friday, Saturday and Sunday, respectively. These improvements are instantaneous and may not reflect the overall performance. Thus, we calculate the resulting improvement of CAD per day in terms of total inter-domain traffic. In particular, CAD reduces the total inter-domain traffic by 64%, 32% and 23% on Friday, Saturday and Sunday, respectively. Both CAD and rCDN result in same MLU of 21%. This is because CAD jointly manages the network and caching resources while fetching or creating representations on-demand. In addition, CAD increases the number of segments at client buffers by up to 14%.

**Impact of Active Distribution.** To shed some lights on the importance of the *active distribution* feature of CAD, we plot in Figure 4.5c the traffic served by active distribution in CAD (i.e., creating representations on demand). This figure shows the difference in inter-domain traffic when CAD employs only cooperative distribution and when it uses both cooperative and active distribution. Specifically, without active distribution, the traffic shown in Figure 4.5c would have been requested from the origin server. In terms of the total inter-domain traffic, the on-demand processing contributes to about 32% of the improvement in inter-domain traffic. In particular, CAD reduces the total inter-domain traffic by 138 Gb compared to rCDN. In Figure 4.5c, the total traffic created on-demand is 45 Gb, which is 32% of the improvement in the inter-domain traffic. For daily inter-
Figure 4.5: Time series of inter-domain traffic, buffer occupancy, and on-demand created traffic over the 3-day experiment period.
domain traffic, active distribution contributes to about 50%, 27% and 35% of the improvements on Friday, Saturday and Sunday, respectively.

**CAD Respects the Maximum Processing Latency.** For every storage capacity setting, we plot the processing latency distribution of CAD to create video representations on-demand as a box plot in Figure 4.6. The box plot is a compact visualization of the three quartiles 25%, 50% and 75%, minimum and maximum values, and the outliers for a given distribution (i.e., the processing latency in our case). For example, when the storage capacity is 1% of the video library size, 25%, 50% and 75% of the created representations on-demand are processed in less than 0.75, 1.5 and 2.75 seconds, respectively. Moreover, the minimum and maximum processing latency are 0.25 and 3.5 seconds. For example, when the storage capacity is 8% of the video library size, 25%, 50% and 75% of the created representations on-demand are processed in less than 0.5, 0.65 and 1.25 seconds, respectively. Moreover, the minimum and maximum processing latencies are 0.25 and 2.4 seconds. The figure shows that CAD strictly respects the maximum allowed processing latency set by the operator (i.e., 5 seconds). It also shows that, as the storage capacity increases, CAD stores and fetches more representations as indicated by the reduction in the latency. This ensures that the end-to-end latency is minimized as the telco-CDN resources allow.

**Impact of Varying Popularity Patterns.** We assess the impact of various popularity patterns of the video representations. We conduct three experiments where we set the plateau parameter of the Zipf-Mandelbrot distribution $q$ to: 1 (highly skewed popularity), 5 (moderate skewness), and 10 (less skewness, i.e., more objects have similar popularity). We set the storage capacity to 10% of the video library size.

Figure 4.7 depicts the averages of the three considered performance metrics across the entire three-day period. The figure shows that CAD consistently outperforms rCDN in terms of amount of inter-domain traffic and buffer occupancy. Figure 4.7a shows the inter-domain traffic results. CAD outperforms rCDN by up to 54% when $q = 10$. This is because decreasing skewness (i.e., increasing
Figure 4.7: Average metrics of CAD and rCDN under different popularity patterns. Storage capacity is 10% of the library size.

$q$) gives CAD the room to fetch and create more representations on-demand inside the telco-CDN. In term of the buffer occupancy at clients, CAD outperforms rCDN by up to 14% when $q = 1$. As $q$ increases (i.e., less skewness), the buffer occupancy decreases because more requests are served by the origin server as Figure 4.7a shows. Finally, the MLU of CAD increases from 22% to 26% when $q = 10$ compared to rCDN as depicted in Figure 4.7c. This is because decreasing skewness allows CAD to fetch and create more representations on-demand. This small increase in MLU does not diminish the gain of reducing the utilization of the more expensive inter-domain links by 54% [105].

**Running Time of CAD.** We measured the running time of CAD, which varied from 46 seconds to 140 seconds to calculate the results of a whole day. The running time varies based on the expected demands for each day. This shows two important aspects. First, CAD is practical and can run on commodity servers. Second, it gives the telco-CDN operator room for reacting upon demand variations and network failures. For example, the operator can reduce the length of CAD period from 24 hours (as in our experiments) to order of tens of minutes.

**Summary of Results.** CAD outperforms the closest work in terms of inter-domain traffic by a wide margin (up to 64%). In addition, CAD has modular functions and improvements. That is, CAD
can be deployed with no on-demand transcoding. As shown in Figure 4.5c, the on-demand creation results in 32% of the improvements in the total inter-domain traffic, the other 68% improvements are due to the careful TE and content distribution done by CAD. That is, if caching sites have processing resources, CAD uses them to improve the inter-domain traffic. Otherwise, CAD will rely on careful TE of network paths.

4.7 Summary

We considered the problem of managing the resources of the emerging telco-CDNs, which are content distribution networks operated by ISPs. We proposed a new algorithm called CAD (short for Cooperative Active Distribution) to solve this problem. The key new ideas in CAD include: (i) jointly optimizing the utilization of the caching resources and the selection of traffic paths within the network, (ii) creating video representations on-demand, and (iii) enabling cooperation among caching sites to serve as much as possible of the multimedia traffic within the ISP. We implemented and evaluated CAD in a Mininet-based emulation network, which carries real traffic using SDN-managed virtual switches. We also implemented caching servers and DASH clients, and used them to serve and request video segments. Our results show that compared to the closest work in the literature, CAD achieves up to 64% reduction in the inter-domain traffic, which is a major cost factor for ISPs. CAD also increases the number of segments in the client buffer by up to 14%, which is an important metric to measure user QoE. Finally, CAD chooses the traffic paths carefully to avoid introducing bottlenecks in the ISP network. These improvements are critical for ISPs and content providers who collaborate, especially with the growth of the viewership of these content providers.
Chapter 5

Stateless Traffic-Engineered Multicast Forwarding

In this chapter, we describe the design of a label-based multicast forwarding system for ISP networks where the controller cannot directly instruct core routers to update their state. We evaluate the proposed system in hardware testbed and simulations to assess its performance.

5.1 Introduction

Recent large-scale Internet applications have introduced a renewed interest in scalable multicast services. Examples of such applications include live Internet broadcast (e.g., Facebook Live and Periscope), IPTV [54], video conferencing [24] and massive multiplayer games [27]. The scale of these applications is unprecedented. For instance, Facebook Live aims to stream millions of live sessions to millions of concurrent users [136, 114]. As another example, BBC reported a peak of 9.3 million TV users watching the Euro 2016 opening, and 2.3 million users watched that event through the BBC Internet streaming service [113]. The amount of live streaming traffic is expected to occupy 17% of the total Internet video traffic by 2022 [1]. These applications thus have restored the need for scalable multicast systems. In particular, to reduce the network load of such applications, ISPs use multicast to efficiently carry the traffic through their networks. For example, AT&T has deployed UVerse and BT has deployed YouView multicast services.

Providers of large-scale live applications require ISPs carrying their traffic to meet target quality metrics or SLAs (service level agreements), especially for popular live multicast sessions. To achieve target SLAs for various customers, ISPs need to carefully manage and direct various traffic flows through their networks. This is usually referred to as Traffic Engineering (TE), and goes well beyond just forwarding the traffic along the shortest paths to destinations. In many cases, the network paths are chosen (or engineered) to meet the requirements of an application/session/customer, and these chosen paths may not necessarily be the shortest paths computed by the routing protocols. For example, latency-sensitive applications require passing through low-delay network paths,
whereas high-volume content distribution applications may need to go through paths with large available bandwidth.

Multicast traffic engineering is the task of finding optimized multicast paths in the network according to a given network objective, such as minimizing the maximum link utilization [57]. Paths of a multicast session form a distribution tree. If a multicast tree is computed to meet traffic engineering objectives (i.e., not merely a minimum spanning tree), we refer to it as a traffic-engineered multicast tree. Different methods have been proposed in the literature to create traffic-engineered multicast trees such as [120, 70, 69, 25].

To implement traffic-engineered trees in the data plane, ISPs need to deploy a multicast forwarding system. However, designing efficient multicast forwarding systems is quite challenging, as it needs to address three main issues: (i) scalability, (ii) efficiency, and (iii) generality. The scalability has been a major issue since the introduction of IP multicast [33]. The main concern is that group management and tree construction protocols (i.e., IGMP [19] and PIM [45]) require maintaining state at routers for each session. In addition, these protocols generate control messages among routers to refresh and update this state. Maintaining and refreshing state for each session impose significant overheads on core routers that need to support high-speed links. Thus, in practice, router manufacturers tend to limit the number of multicast sessions. For example, the Cisco ASR 9000 series can maintain up to 1,024 multicast sessions [2]. Moreover, label-based approaches such as BIER [135] and mLDP [130] lack generality as they only use the shortest paths computed by the routing protocols.

In this chapter, we address the lack of scalable and general multicast forwarding systems for large-scale ISPs. We propose a new label-based approach called STEM (Stateless Traffic-Engineered Multicast) to implement traffic-engineered multicast trees. STEM is scalable as it does not require state at routers and does not impose significant communication overheads. It only updates one (ingress) router when a tree changes. This allows fast adaptation to network dynamics such as routers joining/leaving sessions and link failures. STEM is also general because it can implement any traffic-engineered trees. STEM achieves scalability and generality by moving the forwarding state from routers to labels on packets. Every label in STEM contributes to representing a given tree.

We implemented STEM in a hardware testbed using NetFPGA. Our implementation and experimentation show that STEM can support high-speed links carrying thousands of concurrent multicast sessions. To assess STEM performance in large-scale networks, we compare STEM against a rule-based approach implemented using OpenFlow and a label-based approach [81] in simulations using real ISP topologies with different sizes. Our results show that STEM reduces the number of routers to be updated when a multicast session changes by up to 30X and 10X compared to the rule-based and the label-based approaches, respectively. Moreover, STEM attaches small label sizes compared to the typical packet payload of multicast applications. For example, the label size of 90% of the packets is less than 24 bytes for a real ISP network.
5.2 Related Work

We divide the related works in the literature into two categories: rule-based and label-based.

**Rule-based Approaches.** These multicast approaches require storing forwarding state about sessions at routers. The traditional IP multicast [33] is an example of such approaches. IP multicast is implemented in core routers produced by most vendors. However, it suffers from scalability issues in real deployments [38]. In particular, group management and tree construction protocols (e.g., IGMP [19] and PIM [45]) require maintaining state at routers for each session, and they generate control messages among routers to refresh and update this state. Maintaining and refreshing state for each session impose significant overheads especially on core routers that need to support high-speed links. Thus, in practice, router manufacturers tend to limit the number of multicast sessions. For example, the Cisco ASR 9000 series can maintain up to 1,024 multicast sessions [2]. In addition, IP multicast uses shortest paths and cannot implement traffic-engineered trees.

Recent SDN-based protocols isolate the control plane from the data plane, and can implement traffic-engineered paths. For example, a rule-based approach can be implemented using OpenFlow [82], where the controller installs header-matching rules and actions to forward/duplicate packets. Since OpenFlow stores state at every router along the multicast trees, the total forwarding state at routers grow with the number of sessions. Moreover, the controller has to schedule rule updates when trees change over time in order to guarantee correct forwarding [80, 151].

Li et al. [89] proposed a multi-class Bloom filter (MBF) to support multicast in data center networks. For every interface in a router, MBF uses a Bloom filter to store whether packets of a specific session should be duplicated on that interface. MBF determines the number of hash functions per session based on the prior knowledge of joining events and session size. MBF assumes that the number of sessions is at steady-state. Otherwise, it results in high communication overhead. MBF cannot completely avoid redundant traffic due to the probabilistic nature of Bloom filters.

In these systems, various events, e.g., router joining/leaving and link failures, trigger changing the multicast trees. These changes may require updating the state at many routers, which imposes additional communication overheads. Furthermore, frequent router updates could introduce inconsistency in the network state, especially for large ISPs. To avoid inconsistency, the control plane has to use a scheduling algorithm to gradually update the forwarding rules at routers [80]. Rule update scheduling reduces the network agility (i.e., the ability of the network to quickly react to dynamic changes) as the updated rules take time to be installed/activated at all intended routers; greedily updating rules may result in violating service level agreements [151].

**Label-based Approaches.** STEM is not the first system that moves forwarding information as labels attached to packets. However, prior systems did not concurrently address the aforementioned challenges. For example, early work by Chen et al. [23] proposed a label-based system that attaches link IDs to every packet in a multicast session, and removes unwanted portions of the label as the packet traverses the network. The processing algorithm in [23] requires maintaining state at every router belonging to a multicast session in order to remove the labels, which is not scalable. Later
## Design System Features

<table>
<thead>
<tr>
<th>Design Choice</th>
<th>System</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Scalable</td>
</tr>
<tr>
<td>Rule-based</td>
<td>IP Multicast</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>OpenFlow</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>MBF</td>
<td>Yes</td>
</tr>
<tr>
<td>Label-based</td>
<td>mLDP</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>BIER</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>LIPSIN</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>STEM</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of current multicast forwarding systems.

Works, e.g., mLDP [130], enable multicast in label-switched paths (LSPs). mLDP forwards traffic on the shortest paths and cannot support traffic-engineered trees, and it requires an additional protocol to distribute labels among routers. Recently, works such as BIER [135] encode global router IDs of tree receivers in the label as a bit map. Similar to mLDP, BIER only supports shortest paths. A very recent amendment called BIER-TE [40] attempts to implement traffic-engineered trees inside the network. Routers in BIER-TE need to maintain state to map each bit position in a label to a link attached to a router. To ensure that bit positions for different sessions do not collide, and thus resulting in inefficient forwarding, BIER-TE requires routers to maintain multiple tables (state).

LIPSIN [81] uses a Bloom filter as label to encode global link IDs of a tree. LIPSIN may result in redundant traffic or forwarding loops because of the probabilistic nature of Bloom filters. Thus, LIPSIN requires an additional protocol where downstream routers notify upstream ones if they falsely receive a packet. This protocol imposes additional state and communication overheads on routers. Similar to LIPSIN, the algorithm in [64] encodes link IDs of every tree as integer values, and attaches these values as labels to packets. The algorithm encodes different link IDs with the same integer to reduce the label size, and thus it may result in redundant traffic or forwarding loops.

Segment routing (SR) [46] is a recent proposal to support traffic-engineered unicast flows. It was later extended to support multicast by considering every tree as a segment in the network. It attaches one label containing the tree ID to packets of a session. Routers along the tree maintain a mapping between that label and the output interfaces (i.e., forwarding state). SR and its extensions do not need label distribution protocols such as LDP [10]. However, SR multicast extensions require maintaining state at routers for every tree.

Table 5.1 summarizes the features of current multicast forwarding systems and compares them against STEM.
5.3 System Model and Problem Definition

Multicast services can be used in various scenarios. A common use-case is when a major ISP, e.g., AT&T, manages multicast sessions for its own clients. Clients in this case can be end users in applications such as IPTV and live streaming. Clients could also be caches for content providers such as Netflix, where the contents of such caches are periodically updated using multicast. Another common use-case for multicast services happens when large-scale content providers, such as Facebook and Twitch, partner with ISPs to deliver live streams to millions of users. Our abstract system model, shown in Figure 5.1, supports these and other use cases of multicast.

In particular, our model assumes that the ISP network has data and control planes. The data plane is composed of routers deployed in multiple geographical regions, and these routers are classified as core, ingress, and egress based on their location in the topology. Every router is assigned a unique ID and maintains two data structures: Forwarding Information Base (FIB) and interface list. FIB provides reachability information between routers using shortest paths. The interface list maintains the IDs of all local interfaces. The control plane (referred to as the controller) learns the ISP network topology, shortest paths between all routers, and interface IDs for every router. This is simple to achieve using common intra-domain routing and monitoring protocols.

A multicast session is defined by a distribution tree $T$ that the controller calculates. Note that STEM is orthogonal to the algorithms that calculate traffic-engineered trees [120, 70]. Specifically, for every join/leave event in a session, the controller invokes a multicast update algorithm (e.g., [120]) to calculate a new distribution tree. Edges of the distribution tree may not follow shortest paths computed by intra-domain routing protocols. That is, paths in the tree are chosen to satisfy the objectives of the ISP administrator (e.g., minimize the maximum link utilization). For illustration, in Figure 5.1, although links (3, 6) and (2, 4) are on the shortest paths from the source to $R_1$, the ISP may not want to include them in $T$, because they could be overloaded. Nodes in the distribution tree represent routers, not end users. The root of the multicast tree is referred to as the ingress router, where the multicast traffic of the session is injected. Leaf nodes in the multicast tree are referred to as egress routers, where end users of the multicast session are attached. To join or leave a session, end users notify the corresponding egress router, which notifies the controller. The controller may need to update $T$ accordingly.

The goal of this chapter is to design a scalable, efficient and general multicast forwarding system. That is, given a traffic-engineered multicast tree $T$, we design a system to implement $T$ in the data plane. It is important to note that we are only addressing the problem of efficiently realizing a given multicast tree in the network, which is the core problem in the whole multicast service. Other issues, including computing the tree itself and multicast membership management are outside the scope of this work.

As discussed earlier, current works in the literature do not concurrently address the scalability, efficiency, and generality challenges. In this work, we shed the light on a new point in the design space of multicast forwarding systems, and show that we can design a label-based system
that addresses the aforementioned challenges. The main idea is to completely move the forwarding information for each tree to the packets themselves as labels. Designing and processing such labels, however, pose key challenges that need to be addressed. First, we need to systematically define how the tree forwarding information can be represented in labels. Second, the processing overheads and hardware usage at routers need to be minimized. This is to support many concurrent multicast sessions, and to ensure the scalability of the data plane. Third, when a packet is duplicated on multiple outgoing interfaces, its subset of labels needs to be duplicated as well. If this subset of duplicated labels is not chosen carefully, redundant labels will be added to packets, which increases the communication and processing overheads on routers. Finally, forwarding packets should not introduce ambiguity at routers. That is, while minimizing label redundancy and overheads, we must guarantee that routers will forward packets to and only to the appropriate multicast paths.

5.4 Proposed STEM System

5.4.1 Overview

STEM is a label-based multicast forwarding system that efficiently implements traffic-engineered trees. As shown in Figure 5.1, STEM consists of centralized controller and packet processing algorithm. The controller calculates the distribution tree for a multicast session and creates, using our algorithm in Section 5.4.3, an ordered set of labels \( \mathbb{L} \) to realize this tree in the data plane. As detailed in Section 5.4.2, STEM defines four types of labels; each serves a specific purpose in implementing
the multicast tree in efficient manner. The controller sends the set of labels $L$ to the ingress router of the session, which in turn attaches them to all packets of the multicast session. When a distribution tree $T$ changes (due to link failure or egress router joining/leaving the session), the controller creates a new set of labels and sends them to the ingress router.

The packet processing algorithm, described in Section 5.4.4, is deployed at core routers. It processes the labels attached to packets and forwards/duplicates packets based on these labels. It also attaches different subsets of the labels to the forwarded/duplicated packets such that the subsequent routers can process these labels to implement the distribution tree.

To understand the big picture before delving into the details, we refer the reader to the illustrative example in Figure 5.3. This example implements the distribution tree marked by the dashed lines in Figure 5.1; the details of the calculations will be presented in Section 5.4.6. The ingress router (green diamond) gets a set of carefully structured labels from the controller. The packet processing algorithm at node 1 duplicates packets on its two interfaces. Notice that packets forwarded on interface 1 has a different subset of the labels than the packets forwarded on interface 2. A similar behavior is observed at subsequent routers, where packets are forwarded only on links that belong to the tree and the subsets of labels attached to packets get smaller as we get closer to the leaf nodes of the tree.

### 5.4.2 Label Types in STEM

STEM is a label-based system. Thus, one of the most important issues in STEM is to define the types and structures of labels in order to minimize the communication and processing overheads while still be able to represent general traffic-engineered multicast trees. To address this issue, we proposes the four label types shown in Table 5.2. The first two label types are called Forward Shortest Path (FSP) and Forward Traffic Engineered (FTE). They are used by routers to forward packets on paths that have no branches. The other two label types are Multicast (MCT) and Copy (CPY). The MCT label instructs routers to duplicate a packet on multiple interfaces, and the CPY label copies a subset of the labels. Every label consists of two fields: type and content. The type field is used by routers to distinguish between labels during parsing, and the content field contains the information that this label carries. The size of a label depends on the size of the content field.

We use the example topology in Figure 5.2 to illustrate the concepts used in defining STEM labels. In the figure, solid lines denote tree links and dotted lines denote links on the shortest path from node 1 to node 7. The ISP avoids them because they are over-utilized in this example.

We divide a multicast tree into path segments and branching points. A path segment is a contiguous sequence of routers without branches. A branching point refers to a router that duplicates packets on multiple interfaces. For the example in Figure 5.2, there are three path segments: $\{1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7\}$, $\{8 \rightarrow 9\}$, and $\{10 \rightarrow 11\}$. Routers 7 and 11 are branching points.

The basic label in STEM is FTE, where a router is instructed to forward the packet carrying the label on a specific interface. In many situations, however, packets follow a sequence of routers on the shortest path. For these situations, we define the FSP label, which replaces multiple FTE labels
with just one FSP label. An FSP label contains a global router ID, which instructs routers to forward incoming packets on the shortest path to that router. For example, in Figure 5.2, instead of using two FTE labels for the links \{5 \rightarrow 6\} and \{6 \rightarrow 7\}, STEM uses one FSP label with destination ID set to node 7. In large topologies, FSP labels can achieve significant savings in label overheads.

FTE and FSP labels can represent path segments, but they do not handle branching points where packets need to be duplicated on multiple interfaces. Notice that, after a branching point, each branch needs a different set of labels because packets will traverse different routers. To handle branching points, we introduce the MCT and CPY labels. The MCT label instructs routers to duplicate packets on multiple interfaces using a bitmap of size \(I\) bits, where \(I\) is the maximum interface count per router. The bitmap represents local interface IDs, where the bit locations of the interfaces that the packet should be forwarded on are set to one. Unlike other label types in STEM, the CPY label does not represent a forwarding rule. Instead, it instructs a router to copy a subset of labels (that follows the CPY label) when duplicating packets to a branch without copying all labels. Specifically, consider a router that duplicates packets to \(n\) branches. The MCT label is followed by \(n\) sets of labels to steer traffic in these branches, where every label set starts with a CPY label. The
CPY label of one branch contains an offset of label sizes (in bits) to be duplicated to that branch. For example, in Figure 5.2, router 7 will process an MCT label followed by two CPY labels, one for each of the two branches.

The CPY label content size in Table 5.2 uses the worst case distribution tree. This happens when the tree has the longest height, which is \( O(N) \) and every link is traffic-engineered, where \( N \) is the number of core routers. To further minimize the label overhead, STEM may not create CPY labels if an MCT label duplicates packets to egress routers, or core routers that are directly connected to egress routers. In these cases, routers do not need labels to forward/duplicate incoming packets. We add an additional bit to the MCT label to realize this idea. Hence, the most significant bit of the MCT label is set to one when CPY labels are created. Otherwise, this bit is set to zero. The least significant \( I \) bits contain the interface bitmap.

### 5.4.3 Creating STEM Labels at the Controller

The `CREATELABELS` algorithm, shown in Algorithm 4, runs at the controller to create labels. When the distribution tree \( T \) changes, the algorithm creates a new set of labels \( L \) to reflect this change and sends them to the ingress router, which attaches them to every packet in the multicast session. The controller does not send labels directly to other routers in \( T \).

At high level, the `CREATELABELS` algorithm divides \( T \) into segments and branching points. The algorithm calculates FSP and FTE labels for every segment, and MCT and CPY labels at branching points. The label order is important because it reflects which routers process what labels. The algorithm traverses the core routers of \( T \) in a depth-first search order starting from the core router connected to the ingress router. It keeps track of the router \( r \) that is being visited, and one path segment (segment). Once a router \( r \) is visited, if \( r \) has only one core child (Line 8), this means that \( r \) belongs to the current segment. The algorithm then appends \( r \) to segment, and pushes its child to the stack to be traversed later. For example, the algorithm pushes routers 1, 2, 3, 5 and 6 in Figure 5.2 to segment because each of them has only one child. If \( r \) has no core children or has more than one child, the algorithm calculates labels as follows:

1. **Creating FSP and FTE Labels of a Segment:** The algorithm creates a segment labels when segment ends (i.e., \( r \) has no core children). This happens in two cases. First, when \( r \) is connected to an egress router (e.g., router 9 in Figure 5.2). Second, when \( r \) is a branching point and segment has routers (e.g., router 7 in Figure 5.2). In both cases, the algorithm appends \( r \) to the current segment, calculates FSP and FTE labels for this segment using the algorithm `CALCSEGMENTLBL`, and clears segment.

The pseudocode of `CALCSEGMENTLBL` is shown in Algorithm 5. It takes as inputs a tree \( T \), a path segment segment and the shortest path map \( P \), and calculates the FSP and FTE labels of the given segment. It divides segment into two sub-paths: one that follows the shortest path, and one that does not. It then recursively applies the same to the latter sub-path. Specifically, `CALCSEGMENTLBL` concurrently traverses segment and the shortest path between source and destination. It stops when the traversal reaches a router in segment that does not exist in the shortest path. This
Algorithm 4 Create labels to represent a multicast tree.

Require: $P$: shortest path map

Require: $T$: multicast tree

Ensure: $L$: labels to be sent to the ingress router

1: function CREATELABELS(source)
2:     $L = \{\},$ segment $= \{\},$ stack $= \{source\}$
3:     while (stack.size() > 0) do
4:         $r = stack.pop()$ // a router in $T$
5:         core_children $= T.core_children(r)$ //core routers
6:         children $= T.children(r)$ //core and egress routers
7:         // $r$ belongs to a path segment
8:         if (core_children.size() == 1) then
9:             segment.append($r$); stack.push(children[0])
10:        // $r$ is connected to an egress router
11:        else if (core_children.size() == 0) then
12:            segment.append($r$)
13:            lbls = CALCSEGMENTLBL($T$,segment,$P$)
14:            $L$.push(lbls); segment $= \{\}$
15:        // $r$ is a branching point
16:        else if (children.size() > 1) then
17:            if (segment.size() > 0) then
18:                segment.append($r$)
19:                lbls = CALCSEGMENTLBL($T$,segment,$P$)
20:                $L$.push(lbls); segment $= \{\}$
21:            end if
22:            $\langle mctLbl, cpy \rangle = CREATEMCT(children)$
23:            $L$.push(mctLbl)
24:        // Creating CPY labels
25:        if (cpy) then
26:            for ($c \in$ children) do
27:                br_lbls = CREATELABELS($c$)
28:                offset = CALCLABELSIZE(br_lbls)
29:                $L$.push(CPY(offset)); $L$.push(br_lbls)
30:            end for
31:        end if
32:        end if
33:    end while
34:    return $L$
35: end function
36: return CREATELABELS($T$.source)

means that this router does not follow the shortest path, hence, it adds an FSP label for the previous router. If segment has routers that do not follow the shortest path, CALCSEGMENTLBL adds an FTE label and recursively calls itself using a subset of segment that is not traversed so far. CALC-
Algorithm 5 Create FSP and FTE labels for a path segment.

Require: \( T \): multicast tree

Require: \( \text{segment} \): path segment

Require: \( \mathcal{P} \): shortest path map

Ensure: \( \text{segment}_\text{lbls} \): labels of the path segment

1: function \( \text{CALCSEGMENTLBL}(T, \text{segment}, \mathcal{P}) \)

2: \( \text{segment}_\text{lbls} = \{\} \)

3: \( \text{src} = \text{segment}[0]; \text{dst} = \text{segment}[\text{size} - 1] \)

4: \( \text{sp} = \mathcal{P}[\text{src}][\text{dst}] \)

5: \( i = 0; j = 0 \)

6: while \( (i < \text{segment}.\text{size}() \) and \( j < \text{sp}.\text{size}() \) and \( \text{segment}[i] == \text{sp}[j] \) do

7: \( i += 1; j += 1 \)

8: end while

9: if \( (i > 1) \) then

10: \( \text{fsp}_\text{lbl} = \text{FSP}(\text{segment}[i-1]) \)

11: \( \text{segment}_\text{lbls}.\text{append}(\text{fsp}_\text{lbl}) \)

12: end if

13: if \( (i != \text{segment}.\text{size}() \) or \( j != \text{sp}.\text{size}() \) then

14: \( \text{intf} = T.\text{get_intf}(\text{segment}[i-1], \text{segment}[i]) \)

15: \( \text{fte}_\text{lbl} = \text{FTE}(\text{intf}) \)

16: \( \text{segment}_\text{lbls}.\text{append}(\text{fte}_\text{lbl}) \)

17: \( \text{lbls} = \text{CALCSEGMENTLBL}(T, \text{segment}[i:], \mathcal{P}) \)

18: \( \text{segment}_\text{lbls}.\text{add}(\text{lbls}) \)

19: end if

20: return \( \text{segment}_\text{lbls} \)

21: end function

\( \text{SEGMENTLBL} \) does not generate two consecutive FSP labels. When it calculates an FSP label, it either terminates, or creates an FTE label followed by a recursive call.

For the example in Figure 5.2, the algorithm traverses the segment \( \{1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7\} \) as follows. It finds the first sub-path \( \{1 \rightarrow 2 \rightarrow 3\} \) where the link \((3, 5)\) is not on the shortest path from \(1\) to \(7\). It calculates an FSP label for this sub-path. It then computes an FTE label for the link \((3, 5)\), where the interface ID is \(2\) at router \(3\). Finally, the algorithm recursively calls \( \text{CALCSEGMENTLBL} \) with the sub-path \( \{5 \rightarrow 6 \rightarrow 7\} \) as an input, for which the algorithm creates an FSP label with router ID \(7\).

(2) Creating MCT and CPY Labels at a Branching Point: The \( \text{CREATELABELS} \) algorithm calculates MCT and CPY labels when \( r \) has more than one child. The algorithm calls \( \text{CREatemct} \) that returns MCT label and a boolean value \( \text{cpy} \) indicating whether CPY labels are required. To create an MCT label, \( \text{CREatemct} \) initializes an empty bitmap of width \( I + 1 \) (\( I \) is the maximum interface count per router). For every child \( c \) of \( r \), it sets the bit location in this bitmap that represents the interface ID between \( r \) and \( c \). It checks if CPY labels are needed as follows. If any child \( c \) has at least one core child, this means that this core child needs labels to forward/duplicate packets.
Otherwise, if all children have no other core children, the router $r$ is directly connected to an egress router, or its children are connected to egress routers. Thus, these routers do not need more labels and STEM does not create CPY labels for these branches. For example, at router 7 in Figure 5.2, both core children 8 and 10 have other core children 9 and 11. Hence, two CPY labels are created for the two branches at 7. The algorithm does not create CPY labels at router 11, because its core children 12 and 13 have no other core children.

Recall that a CPY label copies a subset of labels at a specific branch. If CPY labels are needed at the branching point and $r$ has $n$ children/branches, the MCT label is followed by $n$ CPY labels, and every CPY label is followed by labels to forward packets on the corresponding branch. Specifically, the algorithm iterates over the children of $r$. For every child $c$, the algorithm recursively calls CREATELABELS to create labels of the corresponding branch (Line 27). The created CPY label for a branch contains the size of this branch labels in bits to be copied. We calculate this size by accumulating the size of every label type in $br\_lbls$ (Line 28).

**Time and Space Complexities.** The CREATELABELS algorithm traverses each router and link in the distribution tree only once, because the distribution tree is a proper tree with no loops. There are two main functions that are called during the traversal: creating labels for branches, and CALCSEGMENTLBL. In the worst-case scenario, where every router is a branching point, the algorithm needs to create labels for each branch (Lines 25 to 31). Thus, the time complexity of the CREATELABELS algorithm is $O(N^2 + M)$, where $N$ is the number of routers, and $M$ is the number of links. Notice that the CALCSEGMENTLBL algorithm traverses the segment after all routers of that segment is traversed. Thus, it only adds linear overhead to the first term of the time complexity. The space complexity of the CREATELABELS algorithm is $O(N^2D)$, where $D$ is the diameter of the ISP network.

### 5.4.4 Processing STEM Packets in the Data Plane

Ingress routers attach labels $L$ to packets of distribution tree $T$. Core routers process labels attached to multicast packets, and ensure that the actual (IP) payload of a packet is decapsulated before forwarding it to egress routers. Egress routers forward packets to end users.

The proposed packet processing algorithm is deployed at every core router, and it runs on STEM packets. This is done by setting a different Ethernet type for STEM packets at ingress routers. A core router checks the Ethernet type of incoming packets, and invokes the processing algorithm if they are STEM packets. The algorithm forwards/duplicates packets and it remove labels that are not needed by next routers. It also copies a subset of labels at branching points.

The pseudo code of the packet processing algorithm is shown in Algorithm 6. If the packet has no labels, this means the packet reaches a core router that is attached to an egress router. So, the packet is forwarded to that egress router. Otherwise, the algorithm processes labels based on their types as follows:

1. **FSP Label Processing (Lines 4 to 10):** If the FSP label content is not the receiving router ID (Line 6), it means that this router belongs to a path segment. The algorithm then forwards the packet
**Algorithm 6** Process a STEM packet at core routers.

**Require:** \( pkt \): incoming packet in a multicast session

1. **function** \( \text{PROCESS LABELS}(pkt) \)
2. \( \text{if} \ pkt \ \text{has no labels, then forward} \) it to egress router
3. \( L = pkt\.labels; \ type = \text{extract}(L, 2) \)
4. \( \text{if} \ (\text{type} \ \text{is FSP}) \ \text{then} \)
5. \( \text{content} = \text{examine}(L, \text{size(FSP)-2}) \)
6. \( \text{if} \ (\text{content} \ != \ \text{router.ID}) \ \text{then} \)
7. \( \text{forward} \ pkt \ \text{to} \ \text{FIB[content]} \)
8. \( \text{end if} \)
9. \( \text{extract}(L, \text{size(FSP)-2}) \)
10. **function** \( \text{PROCESS LABELS}(pkt) \)
11. \( \text{else if} \ (\text{type} \ \text{is FTE}) \ \text{then} \)
12. \( \text{interface} = \text{extract}(L, \text{size(FTE)-2}) \)
13. \( \text{forward} \ pkt \ \text{to} \ \text{interface} \)
14. \( \text{else} \)
15. \( \text{copy}(pkt\.labels, L); \ pkt\.detach() \)
16. \( mct = \text{extract}(L, \text{size(MCT)-2}) \)
17. \( \text{for} \ (\text{intf} \ \in \ \text{router.interfaces}) \ \text{do} \)
18. \( \text{if} \ ((1 \ll (\text{intf.ID-1})) \ \& \ \text{mct.infts}) \ \text{then} \)
19. \( \text{new}_\text{pkt} = \text{pkt.clone()} \)
20. \( \text{if} \ (\text{mct.has_cpy}) \ \text{then} \)
21. \( \text{extract}(L, 2) \)
22. \( \text{size} = \text{extract}(L, \text{size(CPY)-2}) \)
23. \( \text{new}_\text{labels} = \text{extract}(L, \text{size}) \)
24. \( \text{new}_\text{pkt}.\text{attach(new}_\text{labels}) \)
25. \( \text{end if} \)
26. \( \text{forward} \ \text{new}_\text{pkt} \ \text{to} \ \text{intf} \)
27. \( \text{end if} \)
28. \( \text{end for} \)
29. \( \text{end if} \)
30. **end function**
along the shortest path based on the underlying intra-domain routing protocol without removing the label at Line 7. If the FSP content equals the router ID, this means that the path segment ends at this router, and the packet may have other labels. Thus, the algorithm removes the current label (Line 9) and calls PROCESSLABELS in Line 10 again to process next labels.

(2) FTE Label Processing (Lines 11 to 13): The algorithm removes the label, extracts the local interface ID in Line 12, then forwards the packet on that interface.

(3) MCT Label Processing (Lines 15 to 28): The algorithm first copies the original labels, and removes the labels from the packet. It then extracts the MCT content into mct. The MCT label contains the interface ID bitmap (mct.intfs) and whether it is followed by CPY labels (mct.has_cpy). The algorithm iterates over the router interfaces in ascending order of their IDs in Line 17. It locates the interfaces to duplicate the packet on (Line 18). For every interface included in the MCT label, the algorithm clones the packet. If the MCT label is followed by CPY labels, the algorithm removes the corresponding CPY label, extracts the following labels based on the offset value, and forwards the cloned packet on the corresponding interface.

Time and Space Complexities. The time complexity of the PROCESSLABELS algorithm (in terms of number of processed labels) is $O(I)$, where $I$ is the maximum number of interfaces per router. The algorithm does not require more space than the maintained information by the underlying routing protocols.

The algorithm is to be implemented in hardware, which means that many operations can run in parallel. For hardware implementation on platforms such as FPGAs or ASIC, if a packet needs to be duplicated on $n$ interfaces, the processing of each CPY label can be done independently and in parallel at the output queue of each interface. In addition, the algorithm does not need to process labels followed by each CPY label, and it only needs to know the range of bytes to be included.

5.4.5 Efficiency and Generality Analysis of STEM

The following two theorems prove the efficiency as well as the generality and scalability of STEM.

Theorem 3 (Efficiency). STEM forwards and duplicates packets on and only on links that belong to the multicast tree.

Proof. FSP labels do not result in traffic overheads because: (1) an FSP label with ID $v$ is only added when a router $v$ is traversed by the CREATELABELS algorithm, (2) since every router $v$ is traversed once and only once by the algorithm, only a single FSP $v$ can be added to the labels, (3) in the data plane, the router with ID $v$ removes the FSP $v$ label. Thus, no subsequent routers in the tree can process that label and transmit the packet back to $v$, and (4) since the traversal starts from the source, if a router $u$ precedes router $v$ in the tree, the algorithm guarantees that $u$ is traversed before $v$. Thus, there is no label to forward packets back to $u$.

Similar properties are guaranteed for FTE labels in terms of links. Moreover, attaching multiple FTE labels does not result in traffic overheads. Otherwise, the given distribution tree has loops, or routers do not remove FTE labels.
Finally, in case of MCT and CPY labels, since the CREATELABELS algorithm recursively creates the labels for each branch, the same guarantees apply within a single branch for branching points. In addition, routers in one branch do not forward packets to routers in other branches. This is because (1) the given distribution tree is a proper tree, and (2) every router in a given branch processes the subset of labels duplicated for that branch using the CPY labels.

**Theorem 4** (Generality and Scalability). **STEM supports any multicast tree (i.e., general), and does not impose state overheads at core routers (i.e., scalable).**

*Proof.* It is straightforward to show that STEM is scalable. It does not maintain state at any router, and requires updating one ingress router when a multicast tree changes. STEM is efficient in terms of label overhead as (1) routers remove unneeded labels as the packet traverses the network, and (2) multiple FTE labels can be represented in a single FSP label if the routers are on the shortest path.

The generality of STEM be shown by the construction and definition of STEM labels. Any multicast tree consists of path segments and branching points. FSP and FTE labels can represent any path segment even when all routers do not follow the shortest path (i.e., by using multiple FTE labels). For branching points, MCT labels can duplicate packets on arbitrary interfaces per router, which again may not follow the shortest paths.  

### 5.4.6 Illustrative Example

We present a simple example to illustrate the proposed approach. Figure 5.3 shows the distribution tree of the session in Figure 5.1, where black thick edges indicate the tree links. The dotted lines are the shortest paths to reach R1, which the ISP avoids because they are over-utilized. Router IDs and used interface IDs are shown in the figure. The number of core routers and maximum interface count are 12 and 5, respectively. Thus, the label sizes (in bits) are 8, 8, 6 and 5 for MCT, CPY, FSP and FTE, respectively (Table 5.2).

The controller generates the shown labels using the CREATELABELS algorithm as follows. First, the algorithm creates MCT 1-00011 to duplicate packets on interfaces 1 and 2 at router 1. Since the children 2 and 9 have core children, the algorithm sets the most significant bit in the MCT label to one, and creates two CPY labels for branches A and B. In branch A, the recursive call of Algorithm 4 creates MCT 1-01010 at router 2 to duplicate packets on interfaces 2 and 4. Then, the algorithm recursively calculates labels for branches C and D. For branch C, the algorithm creates a single path segment \( \{3, 5, 6\} \) as follows. First, when it traverses 3 and 5, the algorithm appends them to *segment* because they both have one core child (Line 8, Algorithm 4). When the algorithm reaches 6 (which has no core children), the algorithm appends it to *segment* (Line 11, Algorithm 4) and creates labels for the path \( \{3, 5, 6\} \) (Lines 11 to 14). In this case, since link (3, 5) is not on the shortest path from 3 to 6, the algorithm creates FTE 010 to forward packet on interface 2 at router 3. In addition, the algorithm creates FSP 0110 to forward packets from 5 to 6, because link (5, 6) is on the shortest path between 5 and 6.
We describe the packet processing algorithm at representative routers. The dark labels in Figure 5.3 are the ones that are processed at given routers. Routers 7 and 9 process FSP and forward packets on the shortest path to reach 8 and 10, respectively. Routers 8 and 10 remove corresponding FSP labels because their contents are their router IDs, and then call PROCESSLABELS again. Router 8 forwards the non-labeled packet to the R2, while router 10 removes the MCT label and duplicates the packet to 11, 12 and R3. Routers 11 and 12 forward the non-labeled packet to R3 and R4.

This example shows that the number of labels decreases as packets traverse the tree. For instance, an incoming packet at router 1 has 11 labels. When the packet is duplicated to branch B, only two labels are attached.

### 5.5 Evaluation in a Testbed

We present a proof-of-concept implementation of the proposed multicast forwarding system. We realize that core routers have many different hardware designs and software stacks. The goal of this section is to show that our proposed ideas can be implemented in a representative platform.
Figure 5.4: Setup of our testbed. It has a NetFPGA that represents a core router in an ISP network and runs our algorithm.

(NetFPGA). Implementation in other platforms will be different, but the conceptual ideas are the same.

5.5.1 Testbed Setup

Figure 5.4 shows the setup of our testbed. The testbed has a router representing a core router in an ISP topology that receives and processes packets of concurrent multicast sessions. As shown in the figure, the router is a branching point for all multicast sessions passing through it. This requires duplicating every packet on multiple interfaces, and copying and rearranging labels for each packet. This represents the worst-case scenario in terms of processing for the router, as processing FSP and FTE labels is simple and requires only forwarding packets on one interface and removing a single label. We implemented the router in a programmable processing pipeline using NetFPGA SUME [153], which has four 10GbE ports.

The testbed also has a 40-core server with an Intel X520-DA2 2x10GbE NIC, which is used to generate traffic of multicast sessions at high rate.

Implementation. Our router implementation is based on the reference_switch_lite project for NetFPGAs [104]. This router contains three main modules: input_arbiter, output_port_lookup and output_queues. Our implementation modifies the last two Verilog modules. The first module receives a packet from the input queues, and reads the first label to decide which ports to forward/duplicate the packet on. The second module is executed at every output queue. It detaches labels that are not needed in the following path segments. The second module has input and output queues by its own to store packets during processing. The main design decision for the second module is to process a data beat in two parallel pipelines. The first one is to read a data beat from the module input queue, and remove or rearrange labels as required. The second pipeline is to write the processed data beat to the output queue. We designed this module as a finite state machine that decides when to read/write from/to input and output queues.
Traffic Generation. The 40-core server in our testbed is used to generate traffic of concurrent multicast sessions using MoonGen [42]. We stress STEM by transmitting 10 Gbps of traffic that requires copying and rearranging labels for each packet as shown in Figure 5.4. We attach labels of size 28 bytes to each packet. These labels contain MCT and CPY labels. The MCT label instructs the router to duplicate packets on two ports B and C in Figure 5.4. We monitor and measure the outgoing traffic on port B. The labels instruct STEM to remove 16 non-sequential bytes and keep the other 12 bytes.

5.5.2 Results

We report three metrics that are important in the design of high-end routers: (1) Received throughput, (2) Packet latency, and (3) Resource usage.

Throughput. We transmit labelled packets of concurrent multicast sessions from the traffic-generating server to the router for 60 sec. The main parameter that we control is the packet size, which we vary from 64 to 1024 bytes.

Figure 5.5 shows the throughput of the received multicast traffic from the router. The figure shows that STEM can forward traffic of concurrent multicast sessions at high rates for typical packet sizes. Specifically, for the 1024-byte packet, the received throughput is 9.83 Gbps which is only 1.7% less than the actual bandwidth. We note that multicast is often used in video streaming, which has large packet sizes.

Notice that STEM processes each incoming packet independently. That is, processing packets in STEM does not depend on the number of ports in the router. Hence, its performance still applies for routers with large port density. In addition, our simulation results in Section 5.6 show that the 28-byte label is sufficient to represent a multicast tree in a real ISP network of 84 routers for large receiver density. Thus, the results from our testbed reflect STEM performance in large and real networks.
Latency. We report the packet processing latency at port B in Figure 5.4. We measure the latency by timestamping each packet at the traffic generator, and taking the difference between the current timestamp and the received timestamp at port B. Due to the lack of a specialized hardware equipment, we use the Berkeley Extensible Software Switch (BESS) [60] to timestamp the packets. Since the software layer may add overheads while timestamping and transmitting packets, we compare the latency of STEM processing against the basic IP unicast forwarding in the same testbed. As shown in Figure 5.4, we transmit packets of concurrent multicast sessions at 10 Gbps, where each packet requires copying a range of its labels.

Table 5.3 shows multiple statistics of the packet latency for both STEM and unicast forwarding in µsec when the packet size is 1,024 bytes. The results show that the latency of STEM processing under stress is close to the simple unicast forwarding. For example, the difference of the 95th percentiles of packet latency is only 0.9 µsec.

Resource Usage. We measure the resource usage of the packet processing algorithm, in terms of the number of used look-up tables (LUTs) and registers in the NetFPGA. These numbers are generated by the Xilinx Vivado tool after synthesizing and implementing the project. Our implementation uses 12,677 slice LUTs and 1,701 slice registers per port. Relative to the available resources, the used resources are only 3% and 0.2% of the available LUTs and registers, respectively. Thus, STEM requires small amount of resources while it can forward traffic of thousands of concurrent multicast sessions.

5.6 Evaluation using Simulation

To rigorously analyze the performance of STEM and compare it against the closest approaches in the literature, we have developed a simulator that uses real ISP topologies and considers dynamic and large-scale settings.

5.6.1 Simulation Setup

We implemented a simulator with two components. The first acts as the STEM controller in Figure 5.1. This component receives an egress router event, updates the corresponding multicast tree, and then generates labels using Algorithm 4. The second component simulates the packet process-
ing algorithm in Algorithm 6. The two components of the simulator also implement OpenFlow for comparisons.

We use 14 real ISP topologies [129]. They represent a wide range of topology sizes, where the number of core routers ranges from 20 to 197 per topology. The main parameters we control in the experiments are the receiver density (i.e., the fraction of egress routers in the multicast sessions), number of sessions and number of routers. We create a dataset for every experiment as follows. For every topology, we vary the total multicast sessions count from 100 to 2,000. The sources of multicast sessions are uniformly distributed across the ingress routers. For each session, the maximum number of receivers is randomly chosen to be either 10%, 20%, 30% or 40% of routers in that topology. The session bandwidth is randomly assigned to one of \{0.5, 1, 2, 5, 10\} Mbps values.

We simulate multicast session dynamics such as router joining/leaving as follows. For every router, we generate events based on a Poisson distribution with an average rate of $\lambda$ events per minute, where $\lambda$ is computed by dividing the number of sessions by the number of routers. For every event at a router, we randomly pick a multicast session, and the router randomly joins or leaves the multicast session with probabilities 0.6 and 0.4, respectively. Since STEM does not dictate how multicast trees are computed, we use an algorithm similar to [70] to calculate multicast trees based on link usage.

We compare STEM versus LIPSIN [81], which is the closest label-based multicast forwarding system. LIPSIN encodes the tree link IDs of a session using a Bloom filter. For every link, the LIPSIN controller maintains $D$ link ID tables with different hash values. LIPSIN creates the final label by selecting the table that results in the minimum false positive rate. Since LIPSIN may result in false positives, each router maintains state about incoming links and the label that may result in loops for every session passing through this router. We use the same parameters proposed by LIPSIN: we set $D$ to 8 tables and use five hash functions per link. We set the filter size of LIPSIN to the 99th-percentile of the label size of STEM.

In addition, we implemented a rule-based multicast forwarding system using OpenFlow [82], because rule-based is a general packet processing model that is supported in many networks. The rule-based system installs match-action rules in routers to implement the multicast trees. We refer to this approach by RB-OF in the figures.

We consider four performance metrics: (1) State size: the average number of rules per multicast session at each core router in order to forward/duplicate packets. In the rule-based approach, this is the number of match-action rules. (2) State update rate: the number of messages sent from the STEM, RB-OF, or LIPSIN controller per time unit to routers because of changes in the multicast trees during this time unit. This reflects the overheads associated with updating state at routers. (3) Label overhead: the label size per packet in bytes resulted by STEM. (4) Processing overhead: the number of additional copy operations per packet per router required by STEM. In particular, a STEM router may need to copy a range of labels before duplicating a packet as discussed in Section 5.4. This metric represents the number of additional operations performed by STEM routers compared to RB-OF and LIPSIN.
We report the 95-percentile of all metrics, because it reflects the performance over extended number of sessions or period of time. We repeat every experiment five times and report the average of the 95-percentile across these repetitions.

5.6.2 Results

State Size. Figure 5.6 shows the average state size per multicast session as the density of receivers varies, which corresponds to the number of routers needed to maintain state. The results are shown for a total of 2,000 multicast sessions and for a topology of 197 routers. First, notice that STEM does not require any state at core routers. In contrast, RB-OF needs to maintain state at each router and that state increases with the topology size as well as the density of receivers in each multicast session. For example, the average state size increases from 80 to 130 rules when the receiver density increases from 10% to 30%. Moreover, LIPSIN needs to maintain state at up to 20 routers when the receiver density is 40%.

In another experiment, we varied the number of multicast sessions from 100 to 2,000, and measured the average state size for every router. The state of the rule-based approach linearly increases with the number of sessions. This means that the rule-based approach does not scale when increasing the number of sessions. LIPSIN shows similar trend to the results in Figure 5.6, where this state increases to 145 rules per router. In contrast, STEM does not require maintaining state, and thus it can scale well.

State Update Rate. The key problem with maintaining state at routers is not only consuming their limited memory resources, but also increasing the overheads of updating this state at routers when the multicast tree changes. That is, a router needs to process the update message and modify the corresponding memory location, while forwarding/duplicating packets. We plot the state update rate of STEM, RB-OF and LIPSIN versus the number of sessions for the topology of size 197 routers in Figure 5.7a. The figure shows that the maximum update rate of STEM is only 154 messages/min, while the state update rates of RB-OF and LIPSIN increase to 4000 and 726 messages/min, respectively.
For the same topology, we plot the number of routers to be updated when a single multicast tree changes in Figure 5.7b. The STEM controller needs to only update one (ingress) router when a session changes, which is independent of the topology size. The RB-OF and LIPSIN controllers need to update up to 30 and 10 core routers per tree change, respectively.

This communication overhead impacts the network agility and consistency in the face of new changes. In LIPSIN and rule-based approaches, the network operator has to correctly schedule rule updates to corresponding routers to ensure consistency [80]; greedy rule updates may result in violating TE objectives [151]. STEM avoids the need for complex rule update scheduling algorithms by moving the state to packets, and reduces this rate to one label update per session.

**Analysis of STEM Labels.** We analyze two aspects of the proposed STEM labels: (1) the label size for different scenarios, and (2) the effectiveness of the FSP labels.

The label size per packet in STEM decreases as we move down the multicast tree. This is because routers copy only a subset of labels for every branch using the CPY label. We analyze the label size per packet in terms of how far the packet is from the source and the receiver density, when the number of sessions is 2,000 and the topology size is 84. In Figure 5.8a, we plot the label size versus the number of hops from source, i.e., as a packet traverses the network. The figure shows that the label size in STEM decreases quickly as the packet moves away from the source. For example, the label size is reduced by 18% and 43% after traversing 1 and 5 hops, respectively.

Next, we assess the label size per packet for multiple receiver densities, and present the CDF of the label size in Figure 5.8b. For the 84-router topology and receiver density of 30%, the label size for 90% of the packets is less than 24 bytes. Only 1% of the packets have label size more than 43 bytes. Similar trends are observed for other topologies, but with slightly more sizes for larger topologies. For example, for the largest topology in our dataset (197 routers) and the same receiver percentage of 30%, the label size for 90% of the packets is less than 27 bytes. We note that the label size in STEM is very small compared to the typical packet payload (usually around 1KB) of multicast applications such as video streaming.
Figure 5.8: Analysis of label size and FSP savings of STEM.
Next, we study the importance of the proposed Forward Shortest Path (FSP) label type. Recall that a single FSP label represents multiple routers in a path segment if that segment is on the shortest path. We define the FSP savings as the number of traversed routers per each FSP label. We plot these savings for the topology of size 84 and when the number of sessions is 2,000 as the receiver density increases in Figure 5.8c. The figure shows that FSP labels are effective in representing many routers in the multicast tree. For example, a single FSP label can encode a path segment of length 10 routers when the receiver densities are within 10% to 25%. To show the importance of FSP label across different topologies, we plot the average FSP saving for five sample topologies in Figure 5.8d. We observe consistent savings across the topologies which range from 6 to 10 routers per FSP label.

Analysis of Processing Overhead. In Figure 5.9a, we plot the number of copy operations per packet per router versus the receiver density for multiple topology sizes. The figure shows that the additional work per packet is small. The number of copy operations increases as the receiver density increases because in this case trees have more branches. However, the processing overhead increases slowly. For instance, in the 84-router topology, the average number of copy operations increases from 0.3 to 0.45 per packet per router when the receiver density increases from 5% to 38%. This pattern applies for other topologies as well. That is, STEM routers scale in terms of processing as the network load increases.

We next present the distribution of processing FSP, FTE, MCT and CPY labels per router for the five topologies in Figure 5.9b. The figure shows that the fraction of processed CPY labels (the most expensive) per STEM router across all sessions is small compared to other label types. For example, only 17% of the labels being processed at a STEM router are CPY labels when the topology size is 197. This indicates that CPY labels impose low overheads on STEM routers.

Running Time of STEM Controller. We measured the running time of creating labels per tree update at the controller. We present these results in Figure 5.10 when varying the receiver density for multiple topologies, and show that the additional overhead of creating labels at the controller is small. The running time per tree update varies from 4 msec to 10 msec based on the topology size. For the topology of size 197, the controller spends only 10 msec per tree update to create
the labels for the largest tree. Thus, the proposed label creation algorithm is practical and can run on commodity servers. Moreover, the controller can support frequent tree updates due to session changes and network failures.

**Remarks.** We note that supporting multicast imposes more overheads than unicast packet forwarding. But multicast saves substantial bandwidth and processing compared to unicast. For example, if we were to live stream to millions of concurrent users using unicast, each router would process the same data packet numerous times. Thus, overheads should be considered in the context of the substantial savings achieved by multicast.

## 5.7 Summary

We proposed an efficient, label-based, multicast forwarding system called STEM that implements traffic-engineered trees in ISP networks. Unlike current rule-based systems, STEM does not require additional state at routers. And unlike current label-based systems which forward traffic only on shortest paths, STEM can direct traffic on arbitrary paths in the networks to meet SLAs. The novelty of STEM comes from (1) guaranteed forwarding efficiency, (2) reduced label size, and (3) minimized processing at routers. We evaluated STEM in two setups: hardware testbed and simulations. Specifically, we implemented STEM in a testbed using NetFPGA, and evaluated its performance in terms of throughput, latency and resource usage. Our experiments show that STEM can support thousands of concurrent multicast sessions, and it uses a small amount of hardware resources. To evaluate STEM in large-scale environments, we conducted extensive simulations using real ISP topologies. We compared STEM versus the state-of-art rule-based and label-based approaches. Our results show that STEM reduces the number of routers to be updated when a multicast session changes by up to 30X and 10X compared to the rule-based and the label-based approaches, respectively. In addition, STEM imposes low label and processing overheads.
Chapter 6

Helix: Efficient Multicast Forwarding

In this chapter, we describe the design, analysis and evaluation of a novel multicast forwarding system. The proposed system is designed for ISP networks that enable the controller to update the state at core routers.

6.1 Introduction

In this chapter, we propose a new multicast forwarding system, called Helix, which addresses the multicast forwarding challenges mentioned in Section 1.2.3. Unlike STEM Chapter 5, we design Helix to operate in networks where the controller is able to update the state at core routers. Helix: (i) is general and can be used with any network topology, (ii) does not introduce any loops or false positive packets, and (iii) is scalable as it requires only a small state to be maintained at a fraction of the routers and does not impose significant processing overheads on routers. The key idea of Helix is to systematically split the information about a multicast tree into two parts: constant-size label attached to packets and small state at some routers in the tree.

The information-split architecture in Helix leads to substantial improvements in the data plane performance. First, attaching labels to packets at source (ingress) routers allows quick modifications to trees, while substantially reducing the update rate of state at routers. Second, by only configuring two parameters, Helix can strike a balance between the communication (label) overhead and the memory and processing overheads imposed on routers to handle labels and forward traffic on the multicast tree. Third, packet parsing at routers is simple because the label size is constant, and it has no type-length-value fields. Finally, this architecture can implement traffic-engineered trees in any topology without assuming a specific distribution of multicast sessions.

Helix has both control plane and data plane components that work as follow. In the control plane, Helix uses probabilistic set membership data structures to encode tree links as labels. While these probabilistic data structures reduce the label size, they introduce false positives. A false positive is a link that does not belong to the tree, but the router forwards packets on this link because of the probabilistic nature of the data structure. To address this problem, RLE calculates a set of links that are false positives, and sends them to the routers that handling these links as a state. Helix
minimizes the state by recursively encoding both tree and non-tree links across multiple rounds. In
the data plane, core routers execute a simple processing algorithm on each packet, which utilizes the
information embedded in the labels. In particular, upon receiving a multicast data packet, the router
decides whether to forward/duplicate it on one or more of its interfaces by inspecting the attached
label to the packet and the state maintained at the router, if any.

We have implemented Helix in a NetFPGA testbed to show its feasibility and practicality. Our
experiments show that Helix can easily provide line-rate throughput and it consumes a negligible
amount of the NetFPGA resources. We have also implemented a simulator to compare Helix against
the closest multicast forwarding systems in the literature using real ISP topologies. Our simulation
results show that Helix is efficient and scalable, and it substantially outperforms the closest systems
across all considered performance metrics. We present proofs to show the efficiency of Helix.

To show its feasibility and practicality, we have developed a proof-of-concept implementation
of Helix in a testbed that uses NetFPGA. Our experiments show that Helix can easily provide line-
rate throughput and it consumes a negligible amount of the hardware resources. We have also im-
plemented a simulator to compare Helix against the closest multicast forwarding systems in the
literature using real ISP topologies. Our simulation results show that Helix is efficient and scalable,
and it substantially outperforms the closest systems across all considered performance metrics. For
example, Helix reduces the state per session by up to 18X compared to the closest label-based mul-
ticast forwarding system. Furthermore, compared to a rule-based system implemented in OpenFlow,
Helix decreases the required state per session by up to 112X.

The organization of this chapter is as follows. In Section 6.2, we present the details of the
proposed system, and in Section 6.3 we present our NetFPGA implementation and experiments.
In Section 6.4, we analyze Helix and compare it against others using large-scale simulations. We
conclude the chapter in Section 6.5.

6.2 Proposed Helix System

6.2.1 Overview

Multicast services can be used in various scenarios. A well-known use-case is when a major ISP,
e.g., AT&T, manages multicast sessions for its own clients. Clients in this case can be end users in
applications such as IPTV and live streaming. Clients could also be caches for content providers
such as Netflix, where the contents of such caches are periodically updated using multicast. An-
other common use-case for multicast services happens when large-scale content providers, such as
YouTube, Facebook, Periscope, and Twitch partner with ISPs to deliver live streams to millions of
users.

Figure 6.1 provides a high-level overview of the proposed Helix system. Helix is designed for
ISPs to manage traffic-engineered multicast sessions within their networks. The considered ISP
network has data and control planes. The data plane is a set of routers and directed links connecting
them. Each link \( l \) has a global ID \( l.id \). The control plane is a centralized controller that computes
and updates packet labels and states at routers to forward the traffic of multicast sessions according to the TE objectives of the ISP. Examples of TE objectives include minimizing the maximum link utilization and minimizing the delay.

A multicast session within the ISP has a source and multiple destinations. We refer to the source as the ingress router and the destinations as egress routers. End users (receivers of the multicast traffic) are typically connected to egress routers. Each multicast session has an ID included in all packets belonging to that session. This ID can be inserted, for example, in the MAC destination address. A multicast session is represented by what we call a traffic-engineered tree $T$, which spans the ingress and egress routers of the session. The set of routers that $T$ spans are denoted by $R$. Links of this tree (referred to by $L$) do not necessarily follow the shortest paths; rather they are chosen to achieve the TE objectives of the ISP. Various algorithms can be used to compute traffic-engineered trees such as [120, 70, 69, 25].

The ISP controller and content providers communicate through application-layer APIs to initiate the session and calculate the desired trees. When a multicast session changes (e.g., starts, ends, or user joins/leaves), the corresponding ingress or egress router sends a request to the controller. The controller updates the distribution tree $T$ based on the service level agreements and the session events. Various algorithms can be used to compute customized trees such as [120, 70, 69, 25]. In this chapter, the desired trees are given to the controller and our work is to efficiently realize such trees in the network and support their dynamic changes.

There are two modules in Helix: (i) Creating Labels, which runs at the controller, and (ii) Processing Labeled Packets, which runs in the data plane at individual routers. For every tree $T$, the controller calculates a label to be attached to packets of the session and small state at some routers. It uses a probabilistic set membership data structure (aka filter) to encode the tree link IDs in a relatively small label using hash functions. Since these filters trade off membership accuracy for space efficiency, they may result in false positives, which occur when some links that do not belong to the multicast tree are incorrectly included in the computed filter. False positives waste network resources, overload routers, and could create loops. In Section 6.2.2, we present the details of our method of creating labels while eliminating false positives.

In Helix, various routers process packets as follows. The ingress router of a multicast session attaches the label to packets of that session. Egress routers receive joining/leaving messages from end-users, and send them to the controller to update the multicast trees. Moreover, egress routers detach labels from packets coming from the core network to transmit them to end users. Core routers, on the other hand, use our algorithm (in Section 6.2.3) to forward multicast packets. The algorithm checks labels attached to packets to decide which interfaces to duplicate these packets on. In our design, labels do not change as packets traverse the network. Furthermore, labels are carefully structured so that they can be processed at high speed without overloading routers.
6.2.2 Creating Labels

Designing a label creation algorithm is a challenging task. This is because it needs to achieve the generality, scalability and efficiency requirements. To achieve generality, the proposed algorithm encodes the tree link IDs of every tree using filters, which allow the multicast tree to include links not on the shortest paths. Although these probabilistic filters produce relatively small labels and can be processed efficiently in the data plane, they may result in false positives (i.e., inefficient forwarding). To ensure efficiency, the data plane needs to store these false positives and check them before making decisions. This impacts the scalability of the data plane as it needs to maintain more state.

The key idea to achieve these objectives is to encode both tree and false-positive link IDs inside the label. Thus, the data plane does not need to maintain all false positives. We design an algorithm based on recursive encoding. Our recursive encoding algorithm works in rounds to successively reduce the final state to be maintained at routers. Each round encodes a given set of link IDs based on the outputs of previous rounds, and produces a fixed-size label and an intermediate state. The proposed algorithm carefully controls the inputs of each round to guarantee the correctness of the final outputs. It has provable guarantees on its efficiency as well as time and space complexities. In addition, by parameterizing the number of rounds, the algorithm can control the trade-off between the label size and maintained state.

The algorithm for creating labels takes as input a multicast tree $T$, and produces two outputs: label $H$ and state $S$. Figure 6.2 shows a high-level overview of the proposed algorithm, which we call Recursive Label Encoder (RLE). It recursively calls the BaseEncoder function $K$ times. The
BaseEncoder encodes a given set of link IDs into a label of size $B$ bits and a state $S$. The state can have zero or more entries, and each takes the form $<r, \text{linkID}>$, where $r$ is the router that should maintain this state and $\text{linkID}$ is the ID of the link identified as a false positive during the encoding process.

The inputs to each recursive call of the BaseEncoder are carefully selected to successively reduce the number of entries in the final state as well as facilitate the packet processing at core routers. The label of the session is created by concatenating all intermediate labels produced from the $K$ recursive calls. The state remained after the $K$th call is the final state used in the forwarding decisions.

The BaseEncoder uses a probabilistic set membership data structure (filter) to produce small labels. The BaseEncoder supports any filter that can: (1) add an item to an existing filter, (2) test whether an item exists (potentially with false positives), and (3) avoid false negatives. A false negative happens when a link in the multicast tree $T$ is not represented in the filter. These features are important to create labels and to develop our packet processing algorithm. Examples of such data structures are Bloom [17] and Cuckoo [43] filters.

We define two functions that the filter supports: (i) $H = E_B(l; h)$ to encode an input item $l$ (link ID in our case) into a bit string $H$ of size $B$ bits using hash function $h$, and (ii) $C_B(l, H; h)$ to check whether a given item $l$ belongs to $H$ using hash function $h$. Since the BaseEncoder is recursively called $K$ times and each with a different hash function, we use a set of $K$ independent hash functions denoted by $H_k = \{h_1, \ldots, h_K\}$.

Algorithm 7 lists the pseudo code of the RLE algorithm, which implements the ideas in Figure 6.2. RLE algorithm recursively calls the BaseEncoder $K$ times with different inputs. The main building block is the BaseEncoder (Lines 13 to 25). It encodes every link $l$ in the set of links passed to it using the specified hash function $h$. Then, it calculates the state that needs to be maintained at routers to avoid false positives. It does so by checking all false positive candidates passed in the candidates variable and adding only the entries that collide with the tree links encoded in label to the returned state.

Two important aspects need to be addressed: (i) determining the initial set of false positive candidates, which is the set used in the first round of encoding, and (ii) controlling the input parameters to the BaseEncoder in each of the $K$ encoding rounds. Both directly impact the correctness and efficiency of the proposed label creation algorithm.
Algorithm 7: The Recursive Label Encoder (RLE) algorithm.

Require: \(T\): multicast tree

Require: \(K\): number of filtering rounds

Require: \(B\): number of bits per filter

Require: \(H = h_1, \ldots, h_K\): set of \(K\) hash functions

Ensure: \(H\): label to be attached to the session packets

Ensure: \(state\): state sent to a subset of the core routers

1: function \(RLE(T, K, B)\)
2:   \(H = \{\}\)
3:   \(state = \bot\)
4:     // initial false positive candidates
5:   \(candidates = \text{FindFPCCandidates}(T, R, T, L)\)
6:   \(for (k = 1; k \leq K; k++) do\)
7:     \(\langle H_k, S_k \rangle = \text{BaseEncoder}(state, candidates, B, h_k)\)
8:     \(H = H \cup H_k\)
9:     \(candidates = state\)
10:    \(state = S_k\)
11:  end for
12:  return \(\langle H, state \rangle\)
13: end function

14: function \(\text{BaseEncoder}(links, candidates, B, h)\)
15:   \(label = \text{BitString(size}=B)\)
16:   \(state = \{\}\)
17:   \(for (l \in links) do\)
18:     \(label = label \cup E_B(l.id; h)\)
19:   end for
20: // Calculate state
21:   \(for (l \in candidates) do\)
22:     \(if(C_B(l.id, label; h)) then // false positive\)
23:       \(// add (router ID, link ID) to state\)
24:       \(state = state \cup \{\langle l.src, l.id \rangle\}\)
25:     end if
26:   end for
27:   return \(\langle label, state \rangle\)
28: end function

19: function \(\text{FindFPCandidates}(R, L)\)
20:   \(cand = \{\} // calculated false positive candidates\)
21:   \(L = \{\langle l.dst \rightarrow l.src \rangle for l \in L\} // upstream links\)
22:   \(for (u \in R) do\)
23:     \(for (l \in u.links) do\)
24:       \(if (l \notin \{L \cup L\}) then\)
25:         \(cand = cand \cup \{u \rightarrow l.dst\}\)
26:       end if
27:     end for
28:   end for
29:   return cand
30: end function
Computing Initial False Positive Set. The challenge of finding the initial false positive candidate set is to produce a complete and minimal set. An incomplete set may result in false positives that are not accounted for in the state, which means forwarding multicast packets to links that do not belong to the multicast tree. A non-minimal set increases the state overhead by including state entries that will never be used. For example, choosing a candidate set of all links in the network clearly results in a complete set. However, this set is not minimal and many non-tree links may needlessly collide with the tree links in the calculated label. The key idea of finding this minimal set is that only outgoing links attached to routers belonging to the tree nodes $R$ could be false positives. This is because routers in $R$ do not forward packets on non-tree links since they store these links as state. Thus, if packets of a session would not reach a non-tree router, there is no need to check links attached to that router. For example, in Figure 6.1, link $(a \rightarrow b)$ is removed from the set because router 1 will not forward packets to $a$.

The following theorem states the conditions for the initial false positive candidate set to be complete and minimal.

**Theorem 5.** The initial false positive candidate set is complete and minimal if it is composed of every link $l = (u \rightarrow v) \notin \mathbb{L}$ provided that $u \in R$ and $(v \rightarrow u) \notin \mathbb{L}$, where $R$ and $\mathbb{L}$ are the sets of nodes and links of the multicast tree $T$, respectively.

**Proof.** Recall that the initial set of false positive candidates is the set used in the first round of encoding. We find this set by enumerating all possible combinations of every link $(u \rightarrow v)$ in the network. Then, we derive the sufficient conditions for a link to be a false positive candidate. To achieve this, we check for each link $(u \rightarrow v)$ whether $u$ and $v$ belong to the tree nodes $R$. Table 6.1 lists these combinations with examples from Figure 6.3.

Note that the outgoing links attached to a router $u$ that does not belong to $R$ cannot be false positives (rows 1 and 2 in Table 6.1). This is because a packet can only reach a router $u \notin R$ from a router $p \in R$ by passing through a false positive link $l_{fp}$ attached to $p$. In this case, $p$ should have already stored that the link $l_{fp}$ is a false positive, and it would not forward packets on this link. Hence, a link $(u \rightarrow v)$ where $v \in \text{neighbors}(u)$ cannot be a false positive, and consequently is not a candidate.

Only links connected to a core router in $R$ may be a candidate (rows 3, 4, and 5) as explained below:

1. $u \in R$ and $v \notin R$ (row 3): When a packet reaches a router $u$, this router does not know if $(u \rightarrow v)$ is a non-tree link. This is because this link may collide with the tree links in the first round of encoding. Thus, the BaseEncoder needs to check whether this link collides with the tree links in $H_1$. Thus, $(u \rightarrow v)$ is a candidate.

2. Both $u$ and $v$ belong to $R$. There are two possibilities of links connecting these two routers:

   (a) $(u \rightarrow v) \notin \mathbb{L}$ (row 4). This link can result in forwarding loops because it can be a false positive. Hence it is a candidate, or
Table 6.1: Possible combinations of every link \((u \rightarrow v)\) based on its source \(u\) and destination \(v\) routers. The table shows the conditions to include links in the initial false positive candidate set. The examples are from Figure 6.3.

<table>
<thead>
<tr>
<th>#</th>
<th>(u) (\notin) (R)</th>
<th>(v) (\notin) (R)</th>
<th>Condition</th>
<th>Decision</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(u) (\notin) (R)</td>
<td>(v) (\notin) (R)</td>
<td>None</td>
<td>Not included</td>
<td>(b \rightarrow c)</td>
</tr>
<tr>
<td>2</td>
<td>(u) (\notin) (R)</td>
<td>(v) (\in) (R)</td>
<td>None</td>
<td>Not included</td>
<td>(b \rightarrow 2)</td>
</tr>
<tr>
<td>3</td>
<td>(u) (\in) (R)</td>
<td>(v) (\notin) (R)</td>
<td>None</td>
<td>Included</td>
<td>(1 \rightarrow a)</td>
</tr>
<tr>
<td>4</td>
<td>(u) (\in) (R)</td>
<td>(v) (\in) (R)</td>
<td>((u \rightarrow v) \notin L)</td>
<td>Included</td>
<td>(8 \rightarrow 9)</td>
</tr>
<tr>
<td>5</td>
<td>(u) (\in) (R)</td>
<td>(v) (\in) (R)</td>
<td>((v \rightarrow u) \in L)</td>
<td>Not Included</td>
<td>(4 \rightarrow 2)</td>
</tr>
</tbody>
</table>

In Table 6.1, links that satisfy the conditions in rows 3–5 are complete because there is no other link combination not mentioned in the table. They are minimal because if we exclude any link from rows 3–5, this will result in an incomplete set.

The above theorem allows us to remove many links from the false positive candidate set. For example, links \((2 \rightarrow 1)\), and \((a \rightarrow b)\) in Figure 6.1 cannot be false positives and thus are not considered in the candidate set.

Given Theorem 5, the function \text{FINDFPCANDIDATES} (Lines 26 to 37) in Algorithm 7 calculates the initial false positive candidate set. It initializes an empty set \text{cand}, reverses every link in \(L\) from \((l.src \rightarrow l.dst)\) to \((l.dst \rightarrow l.src)\), and stores this new set in \(\overline{L}\). Note that every non-tree link in \(\overline{L}\) is an upstream link that cannot carry traffic of \(T\). \text{FINDFPCANDIDATES} then iterates over all routers in \(\mathbb{R}\) (Line 29), and for every router \(u\), it adds to \text{cand} the non-tree links that are adjacent to \(u\) and do not belong to \(L\) or \(\overline{L}\).

**Controlling Inputs for BaseEncoder:** The second important aspect in the RLE algorithm is controlling the input parameters to the BaseEncoder to successively reduce the state across the \(K\) encoding rounds, without complicating the process of the algorithm at core routers. As illustrated in Figure 6.2, every round \(k\) of the algorithm produces a label \(H_k\) and an intermediate state set \(S_k\). Each round expects two inputs: (1) links to be encoded \(S_{k-1}\) (solid lines), and (2) false positive candidates \(S_{k-2}\) that may collide with \(S_{k-1}\) in \(H_k\) (dotted lines). These inputs are resulting states from previous rounds, and they are subsets of \(L\) and the false positive candidates. By including \(H_k\) in the final label, the algorithm removes the false positive candidates in \(S_{k-2}\) that are not stored in \(S_k\). That is, in round \(k\), the algorithm decides that some links from that round false positive candidates \(S_{k-2}\) are not needed in the final state \(S_K\). This is because filters in Helix avoid false negatives, and
$H_k$ does not include them. As a result, the algorithm removes these links from the resulting state of this round, and only keeps the links $S_k$ that may be needed in the final state.

Formally, to remove tree and non-tree links from the final state, we extend the definitions of the links to be encoded and the false positive candidates as follows. We denote the tree links $\mathbb{L}$ by $S_0$, and the initial false positive candidate set by $S_{-1}$. For round $k$, the links to be encoded are the resulting state from the previous round $S_{k-1}$. The false positive candidates are the encoded links of the previous round $S_{k-2}$. We define the outputs of round $k$ as:

$$H_k = \text{Encode link IDs in } S_{k-1} \text{ using } E_B, \text{ and}$$

(6.1)

$$S_k = \text{Calculate state using } C_B, H_k, \text{ and } S_{k-2}$$

(6.2)

Generally, RLE uses $K$ labels to remove links belonging to $S_{-1}, S_0, \ldots, S_{K-2}$ from $S_K$. When $K = 1$, RLE removes links from the initial candidate set $S_{-1}$ (i.e., the BaseEncoder). When $K = 5$, RLE uses five labels to remove links belonging to $S_{-1}, S_0, S_1, S_2, S_3$ from $S_5$.

Finally, given the proposed structure of RLE in Figure 6.2, the resulting state in each round contains either tree or non-tree links (but not a mixture of them). To elaborate, in the first round, RLE encodes tree links into $H_1$. It takes $S_{-1}$ as candidates and produces $S_1$ as state. Note that $S_1$ is a subset of $S_{-1}$, hence, the resulting state $S_1$ contains non-tree links only. Similarly, in the second round, RLE encodes $S_1$ into $H_2$ and uses $S_0$ as false positive candidates. Thus, the state $S_2$ contains tree links that is a subset of $S_0$, and collides with $S_1$ in $H_2$. Formally, the following theorem states this property.

**Theorem 6.** In the Recursive Label Encoder (RLE) algorithm, if round $k$ is even, the links encoded in the output state $S_k$ are tree links. Otherwise, these links do not belong to the multicast tree.

**Proof.** We first state two properties about any three consecutive sets. Given $S_{k-2}, S_{k-1},$ and $S_k$, the following two properties hold:

$$S_k \subseteq S_{k-2}, \text{ and}$$

(6.3)

$$S_k \cap S_{k-1} = \phi.$$ 

(6.4)

This means that, for round $k$, the state $S_k$ is a subset of the round candidates $S_{k-2}$. In addition, the intersection of $S_k$ and the round encoded links $S_{k-1}$ is an empty set. The first property holds because, in RLE, the false positive candidate set of a round $k$ is the set $S_{k-2}$. RLE keeps some links from $S_{k-2}$ and removes others. Then it produces $S_k$ as a resulting state. The second property holds because $S_{k-1}$ is the input links to be encoded for round $k$, and $S_k$ is the resulting state from the round candidates. In the first round of RLE, links to be encoded and candidates are non-overlapping sets because they are subsets of $\mathbb{L}$ and the initial false positive candidate set, respectively. Since RLE controls the inputs to Base Encoder by swapping links and candidates, these two sets do not intersect.
We next define $S_k$ in terms of the previous states using this recurrence:

**Base case:**

$k = 0$, $S_0 = \mathbb{L}$ are tree links, and

$k = 1$, $S_1$ are non-tree links.

For $k \geq 2$:

$\forall l \in S_k$, $l \in S_{k-2}$ and $l \notin S_{k-1}$.

For the base case, links in $S_0$ are the tree links by definition. The calculated state $S_1$ are the non-tree links that collide with $S_0$ in $H_1$. The recurrence case (i.e., $k \geq 2$) holds because any link in $S_k$ (1) belongs to the set $S_{k-2}$ (Equation (6.3)), and (2) does not belong to the input set $S_{k-1}$ (Equation (6.4)).

By substituting a given $k$ in this recurrence, we can always check if the set $S_k$ is a subset of the tree links $S_0$ or the non-tree links $S_1$. That is, if $k$ is even, $l \in S_k \subseteq \cdots \subseteq S_0$. Otherwise, $l \in S_k \subseteq \cdots \subseteq S_1$. In addition, a state $S_k$ cannot include tree and non-tree links because of Equation (6.4).

**Time and Space Complexities.** It is straightforward to show that the time complexity of the RLE algorithm is $O(NI + KM)$ and its space complexity is $O(NI + M)$, where $N$ is the number of routers, $M$ is the number of links, $I$ is the maximum number of interfaces per router, and $K$ is number of filtering rounds. The number of routers, interfaces and links in ISP networks are usually in the orders of tens to hundreds. And from our experiments (Section 6.4), $K$ in the range of 4 to 7 results in good performance for most practical multicast trees and network sizes. Recall that the RLE algorithm runs in the control plane and invoked only at the creation of a multicast tree and whenever it changes. Thus, the proposed label creation algorithm can easily be deployed in real scenarios.

### 6.2.3 Processing Labeled Packets

The ingress and egress routers of every multicast session perform simple operations on packets. The ingress router attaches the label $H$ received from the controller to every packet in the multicast session. Egress routers detach labels from packets before forwarding them towards end-users. In this section, we focus on the packet processing algorithm needed by core routers to forward packets of multicast trees.

This algorithm needs to achieve two goals. First, it has to forward packets on and only links belonging to the tree (i.e., efficiency). It does so by exploiting the structure of RLE and the absence of false negatives to make its decisions. Second, it should perform at line rates and consume small amount of hardware resources. We design the algorithm to achieve high performance per packet by executing its operations across links and rounds at same time. In addition, we design the algorithm to use simple instructions, e.g., bitwise operations, that do not require large hardware resources.
Moreover, the algorithm only maintains small data structures (i.e., bit-vectors) to keep intermediate computation results per packet.

Upon receiving a packet at a core router, the algorithm reads both the session ID and the attached label $H$. The algorithm does not share information across different packets. That is, the algorithm checks the label and the state (if any) maintained by the router to decide which of the router’s links belong to the multicast tree. The pseudo code of the proposed packet processing algorithm is shown in Algorithm 8. To efficiently utilize the available hardware resources and accelerate processing, the algorithm makes decisions about all links at same time. Specifically, for each link $l$, the algorithm checks all the $K$ components $H_1, \ldots, H_K$ of the label $H$ as follows. The algorithm searches for the first label component $H_k$ that does not contain the input link ID, $l.id$ (Line 3). If the algorithm finds such label and $k$ is even, the algorithm duplicates the packet on the link $l$. This decision is driven by the design of RLE. Since filters in Helix do not produce false negatives, the algorithm decides (without error) that the input link does not belong to this label component. Moreover, given the structure of RLE, the candidates of the round $S_{k−2}$ contain the link $l.id$, because $H_k$ is the first label that does not contain $l.id$. The algorithm decides whether these candidates are tree or non-tree links based on the value of $k$. If $k$ is even, $k−2$ is even as well. Hence, the links in $S_{k−2}$ are tree links (Theorem 6). If $k$ is odd, these candidates are non-tree links. Checking whether $k$ is even in Line 4 can be done by testing the first bit of $k$, which can be easily implemented in hardware. Notice that routers can use the results of Theorem 6 without storing the actual resulting states $S_1, \ldots, S_{K−1}$.

When $l.id$ exists in all label components, the algorithm needs to check the maintained state $State$. This is because filters in Helix may result in false positives. Recall that this maintained state is a subset of $S_K$ that is calculated by RLE. The algorithm uses the results of Theorem 6 in this case as well. If $K$ is even, the router knows that the links in $State$ are tree links. Thus, the algorithm duplicates the packet if $l.id$ exists in $State[ID]$ (Line 8). On the other hand, if $K$ is odd, the algorithm realizes that the maintained state contains non-tree links. In this case, the algorithm duplicates the packet only if $l.id$ does not exist in $State[ID]$.

The following theorem proves the efficiency of Algorithm 8.

**Theorem 7.** The packet processing algorithm, Algorithm 8, duplicates packets on and only on links that belong to the multicast tree.

**Proof.** The idea of the proof is to use the maintained state and exploit the properties of the RLE and used filters to determine if a link belongs to the multicast tree. To achieve this, we list all cases where a link belongs (or does not belong) to label components.

Given label components, there are three possibilities for a link $l$ attached to a router:

1. this link does not belong to at least one label component $H_k$ but belongs to previous ones $H_1, \ldots, H_{k−1}$,
2. it does not belong to any label component, or
3. it belongs to all label components $H_1, \ldots, H_K$.
In the first case, our proof relies on two properties. First, the candidates of round \( k \) in RLE is \( S_{k-2} \) as stated in Equation (6.2). Second, filters in Helix do not produce false negatives. Let \( H_k \) be the first label where \( l \) does not belong to. The router knows that link \( l \): (1) does not belong to the links in \( S_{k-1} \cup S_k \) (i.e., links that exist in \( H_k \)), and (2) belongs to the candidates \( S_{k-2} \). Otherwise, \( H_k \) would not be the first label where \( l \) does not belong to. This is because RLE removes links from the candidates only if they do not collide in \( H_k \) with the links to be encoded. If \( k \) is even, the router knows that the candidate set \( S_{k-2} \) of this round is a subset of the tree links (Theorem 6). This means that \( l.id \in S_{k-2} \subseteq \cdots \subseteq S_0 \). Thus, the packet is duplicated on \( l \). If \( k \) is odd, the opposite result holds, where \( l.id \in S_{k-2} \) and \( S_{k-2} \subseteq \cdots \subseteq S_1 \). Thus, the packet is not duplicated.

In the second case, if a link does not belong to any label component, this link does not belong to the first label as well. Thus, it is easy to see that this case is a special instance of the first case when \( k = 1 \). Thus, the link belongs to \( S_{-1} \) which are non-tree links.

The third case is when \( l \) belongs to all labels \( H_1, \ldots, H_K \). This means that RLE could not remove this link from all previous states \( S_1, \ldots, S_{K-1} \). And since the filters may result in false positives, it is not conclusive whether \( l \) belongs to the tree links. Thus, we need to look at the content of the maintained state \( S_K \). In this case, we derive the conditions of duplicating packets by using the results of Theorem 6 to decide whether \( S_K \) contains tree or non-tree links as follows:

1. \( K \) is odd. The stored state \( S_K \) is a set of non-tree links (Theorem 6). If \( l.id \notin S_K \), it means that \( l.id \) does not belong to the non-tree links, the packet is duplicated on \( l \). Otherwise, the packet is not duplicated.

2. \( K \) is even. The stored state \( S_K \) is a set of tree links. If \( l.id \in S_K \), then \( l.id \in L \) and the router duplicates a packet on \( l \). Otherwise, \( l \) is a non-tree link, and the packet is not duplicated on it.

\( \square \)

**Implementation Notes.** We perform two further optimizations to execute our algorithm at line rate. First, we do not make routers execute the membership checking function \( C_B \). Instead, each router stores \( K \) bit strings for every link attached to it. These bit strings are calculated by the controller once. We use Bloom filter in our experiments, but other filters can be used as well. Second, we check the \( K \) label components at same time by unfolding the logic inside the loop in Lines 2 to 6. For a representative platform with programmable processing pipeline such as FPGA, the loop can be unfolded across two clock cycles as follows. For each link, the first clock cycle checks whether this link belongs to all label components. For a given link, the second clock cycle finds the first label component where that link does not belong to. This fast implementation comes at a simple cost of storing a small bit-vector when running our algorithm. We discuss the details in Section 6.3.

### 6.2.4 Illustrative Example

We present a simple example in Figure 6.3 to illustrate the various components of Helix. This example uses the topology and multicast tree in Figure 6.1. Figure 6.3a shows a multicast tree that spans the routers numbered 1 to 10. Routers labeled \( a, b, c \) do not belong to the tree. The figure
Algorithm 8 Process labeled packets at core router.

Require: \( l \): link attached to the router
Require: \( H \): Helix label
Require: \( State \): a subset of \( S_K \) stored at the router
Require: \( K \): number of filtering rounds
Require: \( H = h_1, \ldots, h_K \): set of \( K \) hash functions
Ensure: true if duplicating a pkt on link \( l \), else false

// Runs for every link \( l \) attached to the core router
1: function PROCESSLABELED PACKET\((l, H, State, K)\)
2: \hspace{1em} for \((k = 1; k \leq K; k++)\) do
3: \hspace{2em} if (not \( C_B(l.id, H_k; h_k) \)) then
4: \hspace{3em} return \( k \% 2 == 0 \)
5: \hspace{2em} end if
6: \hspace{1em} end for
7: \hspace{1em} if (\( K \) is even) && (\( l.id \in State[ID] \)) then
8: \hspace{2em} return true
9: \hspace{1em} end if
10: \hspace{1em} if (\( K \) is odd) && (\( l.id \notin State[ID] \)) then
11: \hspace{2em} return true
12: \hspace{1em} end if
13: \hspace{1em} return false
14: end function
illustrates the first round of encoding based on the tree links and the initial false positive candidate set. Figure 6.3b shows the labels and states after applying RLE on the same tree when \( K \) is 5. For simplicity, we show the content of a label \( H_i \) as the union of two sets: links encoded in that round \( S_{i-1} \), and false positive links \( S_i \) that collide with these encoded links.

In the first round, RLE encodes the tree links \( S_0 \) into \( H_1 \), and produces state \( S_1 \). Note that \( H_1 \) includes all the links in \( S_0 \) and the links from \( S_{-1} \) that collide with \( S_0 \) in \( H_1 \). Thus, \( S_1 \) is the state of the first round. The second round takes \( S_1 \) as the links to be encoded, and \( S_0 \) as the false positive candidates. This round produces label \( H_2 \) that includes \( S_1 \) and \( S_2 \). Note that in this round, the tree links \((1 \rightarrow 2)\) and \((1 \rightarrow 3)\) (among other tree links) do not collide with any link in \( S_1 \). Hence, RLE decides that these links are not needed in \( S_2 \), because \( H_2 \) does not include them. The final round takes \( S_4 \) as links to be encoded, and \( S_3 \) as false positive candidates. Both non-tree links \((7 \rightarrow c)\) and \((5 \rightarrow 6)\) do not collide with any link in \( H_5 \), and thus they are removed from \( S_5 \).

Notice that the resulting state sets at even rounds \( S_2 \) and \( S_4 \) are tree links, and the resulting state sets at odd rounds \( S_1, S_3 \) and \( S_5 \) are non-tree links.

Finally, we illustrate the decisions made by the packet processing algorithm. First, we show the decisions for some non-tree links. For the link \((7 \rightarrow 8)\), the algorithm finds that it exists in both \( H_1 \) and \( H_2 \), but it does not belong to \( H_3 \) (Figure 6.3b). Thus, the algorithm does not duplicate packets on this link. For the link \((1 \rightarrow a)\), it exists in all labels. In this case, \( K \) is odd and the link exists in \( S_5 \). As a result, the algorithm does not duplicate the packet on this link. For the link \((1 \rightarrow 2)\), the algorithm finds that it exists in \( H_1 \) but not in \( H_2 \). Thus, the algorithm duplicates packets on this link. In the case of link \((3 \rightarrow 6)\), the algorithm finds that it exists in all labels. Since \( K \) is odd and the link does not exist in \( S_5 \), the algorithm duplicates packets on this link.

### 6.3 Evaluation in NetFPGA Testbed

We present a proof-of-concept implementation of the packet processing algorithm of Helix in a testbed. We realize that core routers have many different hardware designs and software stacks. The goal of this section is to show that our proposed ideas can be implemented in a representative platform (NetFPGA). For instance, the line cards of Huawei NetEngine NE5000E core router are implemented using FPGA [144]. Although the implementation of Helix in other platforms will be different, the conceptual ideas are the same.

We use this testbed to show that Helix can sustain line-rate performance with minimal overhead on the hardware resources, and to demonstrate the efficiency of Helix.

#### 6.3.1 Testbed Setup and Algorithm Implementation

**Hardware.** We implemented a router with a programmable processing pipeline using NetFPGA SUME [153]. The router has four 10GbE ports and it is clocked at 156.25 MHz. The testbed also has a 40-core server with an Intel X520-DA2 2x10GbE NIC, which is used to generate and consume traffic at high rate. The router and the server are connected with two SFP+ fiber optic cables. As
shown in Figure 6.4, our implementation represents a core router in an ISP topology. Since the router has four ports, we use one port to transmit packets from the traffic generator to the router. Another port receives packets at the traffic generator after being processed by the router. The remaining two ports are traffic sinks.

**Implementation.** Our implementation is based on the reference_switch_lite project for NetFPGAs [104], which we synthesize using the Xilinx Vivado design suite. This router contains three main modules: input_arbiter, output_port_lookup and output_queues. The input_arbiter module dispatches one packet at a time to the output_port_lookup. The output_port_lookup decides the ports to duplicate packets on by creating a bitmap of the destination ports. The output_queues module forwards the packet based on the bitmap. Our implementation modifies the output_port_lookup module to calculate the bitmap based on the packet processing algorithm as follows. It contains a memory block to store the state. If the controller instructs the router to maintain state for a specific multicast session, our implementation maps the session ID to an I-bit vector, where I is the number of links attached to the router. For every session maintained in memory, if a link \( i \in \{0, \ldots, I-1\} \) is to be stored in this router, the corresponding bit at index \( i \) is set to 1. For every port, the router knows the mapping between this port and link ID. In addition, the router stores \( K \) bit strings each of size \( B \) bits for every port. These
bit strings are the hashed link IDs created by the controller. We use Bloom filter and Murmurhash3 as the hashing function.

When our implementation receives a labeled packet, it parses the $K$ label components by reading the first $K \times B$ bits that follow the Ethernet header. Once the label components are parsed, our implementation runs in three clock cycles (19.2ns) to determine the output ports. In the first clock cycle, the module checks in parallel for each link whether it belongs to every label component $H_k$. Checking whether a link belongs to a label component $H_k$ in Bloom filter is a bitwise-AND operation between the $k$th hashed link ID and $H_k$. This operation is done in one clock cycle. Then, the algorithm stores these results in an $(I \times K)$-bit vector. For every link, the second clock cycle uses the resulting bit vector to detect the index $k$ of the first label where this link does not exist. If one link ID exists in all label components, the algorithm requests a memory read using the session ID. The third clock cycle specifies the final output ports based on the value of $k$ (even/odd) and the state maintained at the router for this session (if any).

Traffic Generation and ISP Topology. The 40-core server is used to generate traffic using MoonGen [42], which allows creating new network protocols such as Helix using Lua language. Since layer-3 protocols are the payload of Helix, we do not generate any layer-3 specific headers. MoonGen can measure the throughput of the received traffic at the server as well. The arrows in Figure 6.4 show the direction of the traffic. We transmit traffic of sessions on one port of the NIC, and receive it through the router on the other port.

We consider a real ISP network topology with 125 routers, chosen from the Internet Topology Zoo [129]. We make our router act as one of these routers. Using this topology, we randomly generate 40 multicast sessions. Every session has a multicast tree covering various routers in the topology. For each session, we use RLE to encode the tree into a label. We set $K$ to 4 rounds and $B$ to 32 bits in our algorithm.

6.3.2 Results

Efficiency of Forwarding Decisions. We first validate the forwarding decisions made by the proposed packet processing algorithm. We wrote a Python script that uses the NetFPGA testing APIs.
Expected 38 7 10
Helix 38 7 10
LIPSIN [81] 38 10 37

Table 6.2: Validation of forwarding decisions made by our packet processing algorithm.

This script transmits one packet on interface nf0 for each of the 40 multicast sessions. It compares the observed forwarding decision of the algorithm against the expected behavior knowing the multicast tree of each session. Our results confirmed that all packets were forwarded as expected, with no false positives (i.e., no packet was forwarded to any link that does not belong to the corresponding multicast session) and no false negatives (i.e., all links that belong to a multicast session received the packet of that session).

We summarize the packet counts on the different interfaces of the router in Table 6.2. For comparison, we also implemented the closest label-based multicast forwarding system in the literature, called LIPSIN [81]. The first row in the table shows the expected number of packets on each interface for the 40 sessions passing through the NetFPGA router, which exactly matches the number resulted by running our algorithm (second row). The third row (obtained by running LIPSIN) indicates that LIPSIN results in many false positives. For example, interface nf3 received 37 packets instead of the expected 10 packets.

**Resource Usage and Scalability.** We measure the resource usage of the packet processing algorithm, in terms of the number of used look-up tables (LUTs) and registers in the NetFPGA. These numbers are generated by the Xilinx Vivado tool after synthesizing and implementing the project. We vary the number of filtering rounds \( K \) from 1 to 6 and plot the numbers of LUTs and registers used by our algorithm in Figure 6.5a. The figure shows that our algorithm utilizes a tiny amount of the available hardware resources. For example, when \( K \) is 6, our algorithm requires 559 LUTs and 1,420 registers, which represent only 0.13% and 0.16% of the available LUTs and registers, respectively. Moreover, increasing \( K \) from 1 to 6 requires only additional 46 LUTs and 42 registers. Thus, our packet processing algorithm scales well as the number of rounds increases.

Next, we analyze the performance of our algorithm as we increase the number of ports from 4 to 1,024. We increase the number of ports by controlling a parameter in our Verilog implementation. Since our router has only four ports, we map each of the additional ports to a physical port in a round-robin fashion. We set the number of filtering rounds \( K \) to 6. The results of this experiment are shown in Figure 6.5b, which shows that Helix scales as we increase the number of ports. For example, for a large router with 1,024 ports, our algorithm uses only 3.4% and 0.87% of the available LUTs and registers, respectively.

**Throughput Measurement.** We show that our packet processing algorithm can easily handle packets at line rate. In this experiment, we use MoonGen to generate labeled packets of multicast traffic.
at 10 Gbps. We transmit these packets from the traffic-generating server to the router on interface nf0 for 60 sec. We use MoonGen to count the number of packets per second received at each of the other interfaces of the router. We vary the packet size from 64 to 1,024 bytes. We run the experiment five times for every packet size and compute the average across them. In Figure 6.5c, we compare the average number of input packets per second to the router (received on interface nf0) against the average number of packets per second observed on one of the output interfaces (interface nf1); the results for other interfaces are the same. The figure shows that the numbers of transmitted and received packets per second are the same (i.e., no packet losses). We plot the achieved throughput in the same figure. The figure shows that our algorithm can sustain the required 10 Gbps throughput for all packet sizes.

**6.4 Evaluation using Simulation**

In this section, we compare Helix against the closest approaches in the literature in large-scale simulations. And we analyze the effect of varying various Helix parameters.
6.4.1 Setup

We implemented a Python-based simulator that acts as a what-if scenario analysis tool. The simulator allows us to evaluate the performance of different multicast forwarding systems in large setups in terms of label size, topology size, receiver density and number of sessions. The core component of the simulator implements the Helix controller. It receives an event such as a router joining/leaving a session, updates the corresponding multicast tree, and then generates labels and states based on the used algorithm.

**Performance Metrics and Control Parameters.** We consider two main performance metrics:

1. *Session state:* number of routers that need to maintain state for a session.
2. *Number of messages per tree update:* number of messages sent to routers by the controller when a multicast tree is updated to modify the state at these routers.

We report the 95-percentile of these metrics, because it reflects the performance over extended number of sessions or period of time. We repeat every experiment five times and report the average of the 95-percentile of every metric across these repetitions. We also comment on the size of the state maintained by a router.

The main parameters we control in the experiments are:

1. *Label size:* We vary the label size from 48 to 128 bytes.
2. *Receiver density:* It is defined as the number of routers that join a session divided by the total number of routers. We vary the maximum receiver density from 10% to 40%.
3. *Topology size:* We use 14 real ISP topologies from the Internet Topology Zoo datasets [129]. They represent a wide range of ISPs, where the number of core routers ranges from 36 to 197, and the number of links ranges from 152 to 486.

**Systems Compared Against.** We compare Helix versus the closest label-based multicast forwarding system, which is LIPSIN [81]. LIPSIN encodes the tree link IDs of a session using one filter. For every link, the LIPSIN controller maintains \(D\) link ID tables with different hash values. LIPSIN creates the final label by selecting the table that results in the minimum false positive rate. Every router maintains \(D\) tables for every link attached to it. Since LIPSIN may result in false positives, each router maintains state about incoming links and the label that may result in loops for every session passing through this router. To ensure fairness, we use the same parameters proposed by LIPSIN: we set \(D\) to 8 tables and use five hash functions per link. For both Helix and LIPSIN, we use Bloom filters and Murmurhash3 hashing functions. When Helix uses \(K\) filtering rounds and \(B\) bits per round, LIPSIN encodes the links in a label of size \(K \times B\) bits. Thus, both systems use the same label size.

In addition, we implemented a rule-based multicast forwarding system using OpenFlow [82], because rule-based is a general packet processing model that is supported in many networks. The rule-based system installs match-action rules in routers to implement the multicast trees.

**Session Dynamics.** For every topology, we simulate 2,000 dynamic multicast sessions for 12 hours, where receivers join and leave sessions over time. The sources of multicast sessions are uniformly
distributed among the routers. For each session, the maximum receiver density is randomly chosen to be either 10%, 20%, 30% or 40% of routers in that topology. The session bandwidth is randomly assigned to one of \{0.5, 1, 2, 5, 10\} Mbps values.

We simulate multicast session dynamics such as router joining/leaving as follows. For every router, we generate events following a Poisson distribution with an average rate of $\lambda$ events per minute, where $\lambda$ is computed by dividing the number of sessions by the number of routers. For every event at a router, we randomly pick a multicast session, and the router randomly joins or leaves the multicast session with probabilities 0.6 and 0.4, respectively. Since Helix does not dictate how multicast trees are computed, we use an algorithm similar to [70] to calculate multicast trees based on link utilization.

### 6.4.2 Comparison against LIPSIN

**Label Size.** We first analyze the impact of varying the label size on the session state and number of messages per tree update. As illustrated in Figure 6.6 for three representative topologies, our results show that Helix requires maintaining state at fewer routers compared to LIPSIN, and it eliminates the state completely in many cases. There are two points to be observed from the figure. First, Helix achieves better performance using the same label size. For example, when the label size is 48 bytes, 95% of the sessions in LIPSIN need to store state at 18, 32, and 48 routers for the three topologies, respectively. Whereas for the same label size, Helix requires state at 1, 5, and 19 routers for the three topologies, respectively. That is, Helix achieves at least 2.5X and up to 18X reduction in the state compared to LIPSIN. Second, Helix eliminates the need to maintain state with much smaller label sizes compared to LIPSIN. For example, when the topology size is 158 routers and the label size is 64 bytes, Helix does not require any router to store state. LIPSIN, on the other hand, maintains state at 17 routers for the same case. A side benefit of reducing the number of routers maintaining state is reducing the number of messages per tree update sent from the controller (figures are omitted). For example, for topology of size 158 routers and label size of 64 bytes, LIPSIN sends 19 messages to update each session, whereas Helix sends only 1 message. Reducing number of messages improves network agility in case of routers joining/leaving sessions and network failures.

There are two positive consequences of reducing the number of routers that maintain state. First, Helix reduces the total required state at routers. For the 158-router topology, the routers in LIPSIN maintain 763, 338, and 210 entries when using label sizes of 48, 64, and 80 bytes, respectively. In addition, LIPSIN cannot eliminate total state even when using a label size of 128 bytes. For the same topology, routers in Helix maintain only 80 and 3 entries when using label sizes of 48 and 64 bytes, respectively. Routers in Helix do not maintain state when using a label size of 80 bytes.

The second benefit of reducing the number of routers maintaining state is reducing the number of messages per session update sent from the controller. When the topology size is 158 routers and label size is 48 bytes, 95% of the session updates trigger 5 and 37 messages to be sent for Helix and LIPSIN, respectively. When increasing the label size to 64 bytes, Helix and LIPSIN send 1 and
19 messages to update each session, respectively. Reducing number of messages improves network agility in case of routers joining/leaving sessions and network failures.

**Receiver Density.** We next study the the impact of the receiver density on the session state and the number of messages per tree update when the label size is 64 bytes. Figures 6.7a to 6.7c show that the session state of LIPSIN increases linearly as the receiver density increases. On the other hand, the session state in Helix increases at slow rate with increasing the receiver density. For the topology of 125 routers, for example, Helix does not result in state at any router. LIPSIN, however, requires maintaining state even when the receiver density is 10%. For the topology of 158 routers, Helix does not maintain state at any router for receiver density up to 22%. For the largest topology, Helix reduces the session state by up to 15X compared to LIPSIN. As a result of reducing the session state, Helix reduces the number of message per tree update by up to 19X, 20X and 16X for the three sample topologies, respectively.

**False Positives and Loops.** Finally, we analyze the false positive overheads of LIPSIN; Figure 6.8 shows the overheads for three sample topologies as the label size increases. We calculate this overhead by dividing the number of sessions that are falsely passing through a link by the exact number of sessions that should pass through the same link. The figure shows that LIPSIN imposes large overheads due to false positives. For example, when the topology size is 158 routers and label size
Figure 6.7: Impact of receiver density on the session state for Helix and LIPSIN. The label size is 64 bytes.
Figure 6.8: LIPSIN false positive overhead in three topologies. Helix does not produce any false positives.

is 64 bytes, LIPSIN results in a false positive rate of 37%. For the same topology, LIPSIN needs to increase the label size to 128 bytes to eliminate this overhead. These false positives do not only impose redundant traffic, but they also can introduce forwarding loops. For the same topology and label size, if LIPSIN routers do not store state as well as additional per-packet information to detect loops, our simulation shows that there would be 250 sessions with loops (i.e., 12% of sessions).

Helix does not incur these overheads because it eliminates false positives by storing state at routers, and it reduces this state for the same label size by using RLE. For example, for the same topology and label size of 64 bytes, Helix does not maintain state for any session as shown in Figure 6.6b.

### 6.4.3 Comparison against OpenFlow

We compare Helix versus a rule-based approach implemented using OpenFlow. Since OpenFlow does not use labels, we only analyze the impact of receiver density on the performance. We use three label sizes for Helix: 64, 80 and 96 bytes.

Figure 6.9a depicts the session state for one sample topology of 158 routers. The figure shows that Helix outperforms OpenFlow for all receiver densities. For example, when the receiver density is 38%, the 64-byte label results in storing state at only up to 2 routers (1.3% of the routers) for 95% of the sessions. Using 80- and 96-byte labels do not result in any state for any receiver density. For the same topology, OpenFlow requires installing match-action rules at up to 120 routers when the receiver density is 38%. Moreover, in the topology of 197 routers, Helix requires only up to 9% of the routers to maintain state when the label size is 64 bytes. However, OpenFlow maintains state at 72% of the router. This means that Helix scales well compared to rule-based approaches when the expected number of receivers increases.

Increasing the session state at OpenFlow routers not only consumes their limited memory resources, but also increases the communication overhead when updating this state (i.e., when a session is updated). Figure 6.9b depicts the number of messages per tree update for Helix and OpenFlow systems for the sample topology. The figure shows that OpenFlow requires sending 36 mes-
sages to update state when the receiver density is 10%. For the same receiver density, Helix sends only one message to update the label at the ingress router. When the receiver density is 38%, the number of messages of Helix is only 3 messages when the label size is 64 bytes. OpenFlow, on the other hand, requires sending 22 messages to update state at corresponding routers. Moreover, for the largest topology, the Helix label size of 64 bytes requires less than 5 messages per update when the receiver density is 20% (39 receivers out of 197), and up to 20 messages when the receiver density is 38% (75 receivers). When increasing the label size to 80 bytes, Helix can reduce the number of messages to 4 per update for the receiver density of 38%. On the other hand, OpenFlow requires sending 25 messages per update for the same receiver density.

**Summary of the Results.** Our simulation results show that:

- Helix reduces the session state by up to 18X compared to LIPSIN.
- Helix is able to totally eliminate the required state for all sessions in all topologies using only a label of size 80 bytes, while LIPSIN needs to maintain state even with a label of more than 128 bytes.
- Compared to rule-based systems, Helix scales well in terms of session state as increasing the receiver density and topology size.

### 6.4.4 Analysis of Helix Parameters

Choosing working ranges for Helix parameters is important in order to achieve the expected performance gains. We study the impact of choosing $B$ and $K$ on the session state for six topologies of different sizes. In Figure 6.10, we show the session state for one of the topologies (158 routers) for six values of $K$ while increasing $B$ from 2 to 16 bytes. We can draw three conclusions from the results.

First, choosing a proper value for $B$ is more important than increasing $K$. For example, in Figure 6.10, increasing $K$ to 8 when $B$ is small (i.e., 2–4 bytes) does not reduce the session state. This is because small filters cannot encode tree and non-tree links without resulting in many false posi-
Figures 6.10: Impact of filter size $B$ on the session state for multiple values of $K$.

tives. Second, once $B$ is properly chosen, increasing $K$ systematically reduces the session state. In the same figure, when $B$ is 8 bytes, increasing $K$ from 3 to 6 reduces the session state from 40 to 12. Setting $K$ to 8 eliminates the session state.

Finally, choosing good ranges for $B$ and $K$ depends on the topology size. For small topologies (35–75 routers), $B$ can be set to 4–6 bytes, and $K$ to 4–6 rounds. For medium topologies (75–150 routers), $B$ can be set to 8–12 bytes, and $K$ to 5–7 rounds. For large topologies (150+ routers), we can set $B$ to 12–16 bytes, and $K$ to 5–7 rounds. For a given label size, choosing $B$ within these ranges results in better performance than smaller values with large $K$. For example, in the 158-router topology, setting $B$ to 6 bytes and $K$ to 8 results in maintaining state at 38 routers per session. However, setting $B$ to 12 bytes and $K$ to 4 reduces the session state to 4.

6.5 Summary

Recent large-scale live broadcast applications have introduced a renewed interest in designing efficient multicast services. Current multicast systems cannot implement general multicast trees, do not scale in terms of state maintained at routers, or may introduce loops and redundant traffic. We designed, implemented, and evaluated a new multicast forwarding system, called Helix, which is general (can be used with any topology), scalable (minimizes the size and update frequency of the state maintained at routers), and efficient (does not introduce loops or forward packets on links not belonging to the multicast tree). Helix has control plane and data plane components. In the control plane, Helix uses a Recursive Label Encoder (RLE) algorithm to encode multicast trees into labels. RLE uses probabilistic set membership data structures to encode tree links as labels. While these probabilistic data structures reduce the label size, they introduce false positives. To address this problem, RLE calculates a set of links that are false positives, and sends them to the routers that handle these links as a state. RLE minimizes this state by recursively encoding both tree and non-tree links across multiple rounds. This allows RLE to encode more information in the label. In the data plane, core routers execute a simple processing algorithm on each packet, which utilizes the information embedded in the labels by RLE. We analytically proved the efficiency of Helix. In
addition, we demonstrated the practicality of Helix by implementing it in a testbed and showing that it can provide line-rate throughput and it consumes a small fraction of the hardware resources. We have also compared Helix against the closest multicast forwarding systems in the literature using large-scale simulations with real ISP topologies. Our simulation results showed that Helix can achieve significant performance gains over the other systems. For example, Helix reduces the state per session by up to 18X compared to the closest label-based multicast forwarding system. In addition, Helix eliminates the required state for many sessions using smaller labels across different topologies. Furthermore, compared to OpenFlow, Helix decreases the required state per session by up to 112X.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

Streaming multimedia content at large-scale to concurrent users introduces multiple challenges for content providers and ISPs. When streaming immersive videos at large scale, content providers face multiple challenges such as the heterogeneous user interactivities and varying network conditions. Moreover, large content providers require ISPs that carry their video traffic to meet certain service level agreements. Thus, traffic flows of these multimedia sessions need to be carefully engineered through the ISP network in order to satisfy various TE objectives. Achieving the TE objectives for multimedia sessions impacts how videos are stored at caching sites, and how live sessions are distributed to concurrent users. For ISPs, this increases the complexity of managing the network and system resources as well as implementing traffic-engineered systems in the data plane.

In this thesis, we addressed the challenges related to streaming and distributing multimedia content to large number of users. Specifically, we developed algorithms and systems to: (1) adaptively stream multiview videos over HTTP, (2) distribute multimedia content to caching sites inside telco-CDNs while satisfying predefined TE objectives, and (3) deliver traffic-engineered flows of live multimedia sessions through multicast inside ISP networks.

Adaptively streaming multiview videos over HTTP is more challenging than streaming single-view videos, because the system needs to support user heterogeneous interactivities as well as handles the network dynamics. Despite the recent interest in the generation of multiview videos, very few works considered designing rate adaptation algorithms for such complex videos. In Chapter 3, we addressed this problem and presented a novel client-based multiview adaptive streaming over HTTP algorithm (called MASH). MASH achieves high rendering quality, smooth playback and efficient bandwidth utilization by modeling and supporting user interactivities. Specifically, MASH constructs and combines local and global view switching Markov chains to weigh the importance of views in the video. We showed that these models converge and impose low overheads on the client and server. We presented a new buffer-based approach to request segments based on the relative importance of different views and the current network conditions.
We implemented the proposed MASH algorithm and compared it against the rate adaptation algorithm used by YouTube to stream multiview videos. Our extensive empirical evaluation showed that MASH substantially outperforms the YouTube algorithm in terms of rendered quality, number of buffering events, and prefetching efficiency. In addition, our results showed that MASH: (i) is scalable as it does not overload the server, (ii) achieves fairness across concurrent streaming sessions, and (iii) renders smooth and high quality even in presence frequent view changes.

Traditional CDNs are often unaware of the current network state of the ISPs. If they could accurately estimate this varying state, they still cannot control the network paths taken to carry the traffic. The difficulties of enabling ISP and CDN collaboration to deliver content, and the potential business opportunities motivated major ISPs to deploy and manage CDNs inside their networks (referred to as telco-CDNs). In Chapter 4, we considered the problem of managing the resources of these emerging telco-CDNs. We proposed a new algorithm called CAD to solve this problem. The key new ideas in CAD include: (i) jointly optimizing the utilization of the caching resources and the selection of traffic paths within the network, (ii) creating video representations on-demand, and (iii) enabling cooperation among caching sites to serve as much as possible of the multimedia traffic within the ISP.

We implemented and evaluated CAD in a Mininet-based emulation network, which carries real traffic using SDN-managed virtual switches. We also implemented caching servers and DASH clients, and used them to serve and request video segments. Our results show that compared to the closest work in the literature, CAD achieves up to 64% reduction in the inter-domain traffic, which is a major cost factor for ISPs. CAD also increases the number of segments in the client buffer by up to 14%, which is an important metric to measure user QoE. Finally, CAD chooses the traffic paths carefully to avoid introducing bottlenecks in the ISP network. These improvements are critical for ISPs and content providers who collaborate, especially with the growth of the viewership of these content providers.

The online watching of live videos over the Internet is becoming common in many scenarios such as Facebook Live, Periscope and BBC Sport. Delivering live multimedia sessions impose significant overhead on the network usage. Thus, the use of multicast is crucial to alleviate the load of increasing number of live sessions. Implementing multicast trees in the data plane is hard due to lack of scalability in current multicast forwarding systems. In addition, these systems cannot easily adapt to the dynamics of live sessions in terms of routers joining/leaving, or the dynamics of the network such as link failures.

In this thesis, we proposed a label-based multicast forwarding approach, where the multicast tree is represented as a set of labels. Our approach consists of two components: centralized controller and packet processing algorithm deployed at core routers. The controller computes labels for every multicast session, and instructs ingress routers to attach them to packets in the session. The packet processing algorithm processes the labels to forward/duplicate packets. We considered two ISP network deployments based on whether the controller can directly instruct ISP core routers to update their state.
In Chapter 5, we considered ISP networks that do not allow the controller to update state at core routers. We proposed an efficient, label-based, multicast forwarding system called STEM that implements traffic-engineered trees in ISP networks. STEM moves the forwarding state of a multicast tree from routers to labels on packets. As a result, the controller does not need to update core routers when a multicast tree changes. STEM is scalable because it does not require state at routers. It also does not impose significant communication overheads, because it only updates one (ingress) router when a tree changes. This allows fast adaptation to network dynamics such as routers joining/leaving sessions and link failures.

We introduced multiple ideas in the design of STEM to reduce the overall label and processing overheads. For example, STEM strikes a balance between label overhead on packets and parsing overhead at routers. In particular, every label type has a fixed size to minimize the total label overhead while achieving fast label processing. In addition, STEM is designed to require routers to examine only a limited number of labels to forward and duplicate packets.

We evaluated STEM in two setups: hardware testbed and simulations. Specifically, we implemented STEM in a hardware testbed using NetFPGA. Our implementation and experimentation show that STEM can support high-speed links carrying thousands of concurrent multicast sessions. To assess STEM performance in large-scale networks, we compared STEM against a rule-based approach implemented using OpenFlow and a label-based approach [81] in simulations using real ISP topologies with different sizes. Our results show that STEM reduces the number of routers to be updated when a multicast session changes by up to 30X and 10X compared to the rule-based and the label-based approaches, respectively. Moreover, STEM attaches small label sizes compared to the typical packet payload of multicast applications. For example, the label size of 90% of the packets is less than 24 bytes for a real ISP network.

The second ISP network deployment we considered is when the controller is able to update the state at core routers. In Chapter 6, we designed, implemented, and evaluated a novel multicast forwarding system, called Helix, which is general (can be used with any topology), scalable (minimizes the size and update frequency of the state maintained at routers), and efficient (does not introduce loops or forward packets on links not belonging to the multicast tree). In the control plane, Helix uses a Recursive Label Encoder (RLE) algorithm to encode multicast trees into labels. RLE uses probabilistic set membership data structures to encode tree links as labels. While these probabilistic data structures reduce the label size, they introduce false positives. To address this problem, RLE calculates a set of links that are false positives, and sends them to the routers that handle these links as a state. RLE minimizes this state by recursively encoding both tree and non-tree links across multiple rounds. This allows RLE to encode more information in the label. In the data plane, core routers execute a simple processing algorithm on each packet, which utilizes the information embedded in the labels by RLE. We analytically proved the efficiency of Helix.

We demonstrated the practicality of Helix by implementing it in a testbed and showing that it can provide line-rate throughput and it consumes a small fraction of the hardware resources. We have also compared Helix against the closest multicast forwarding systems in the literature
using large-scale simulations with real ISP topologies. Our simulation results showed that Helix can achieve significant performance gains over the other systems. For example, Helix reduces the state per session by up to 18X compared to the closest label-based multicast forwarding system. In addition, Helix eliminates the required state for many sessions using smaller labels across different topologies. Furthermore, compared to OpenFlow, Helix decreases the required state per session by up to 112X.

We implemented the two multicast forwarding systems in a representative platform (i.e., NetFPGA) to show their practicality. To implement these systems, router manufacturers would ideally ship their routers with specialized hardware components to implement them. We showed that these new hardware components would impose negligible overheads in terms of hardware resources. Meanwhile, ISPs can still benefit from our systems by imposing small overheads at the programmable ASICs that are widely deployed at current core routers. We outline how the packet processing algorithms of the proposed systems can be implemented in these routers.

The main operation of STEM is similar to popping a label in various network protocols such as MPLS and Segment Routing. These operations are already included in the OpenFlow standard and can be easily implemented using P4 programming language. In this case, the ISP administrator would install a fixed number of rules inside their routers to match a STEM label to a rule which removes that label and forwards/duplicates the packet.

Helix can be realized in P4-supported routers by implementing the algorithm in three stages. In the first stage, the router would check if a link is encoded in a label component by using wildcard matching of the hashed link ID with that label component. Moreover, the session address is matched against the state maintained at the router (called a table in P4). The matching results are maintained in metadata fields, which are supported in the current P4 standard and are used to carry internal information of a packet over different processing stages. For each link, the corresponding metadata will be matched again against a set of rules representing different first set bit locations in that metadata. Since the number of label components $K$ is fixed, we can maintain a constant number of these rules. In the last stage, the router duplicates a packet on a specific link based on the value of the matched set bit location (i.e., even or odd) as well as the content of the maintained state.

The proposed systems, STEM and Helix, are suitable for large-scale ISPs that support multicast even if a single ISP spans multiple geographical regions. In some cases, popular multicast sessions may span multiple ISPs. In this case, Automatic Multicast Tunneling [18] can be used to deliver multicast traffic across different ISPs. Akamai has showed a proof-of-concept implementation of this approach [5] to transmit multicast traffic over multiple carriers.

### 7.2 Future Work

The work discussed in the thesis can be extended in multiple directions. In Chapter 3, we described a rate adaptation algorithm for streaming multiview videos, where our algorithm models user interactivity using Markov model. To support 360-degree and virtual reality videos, the algorithm should
consider the small buffers of their tiles [143]. In addition, since the user watches many tiles in the
same view, the joint probability distribution of tile watching needs to be modelled.

In Chapter 4, we considered the joint traffic engineering and content distribution problem in
telco-CDNs. We leverage the practice that content providers (e.g., Netflix and Hulu) estimate the
popularity of their video library in order to distribute the content. One possible extension for our
work is to support user-generated videos.

In Chapters 5 and 6, we proposed multicast forwarding systems to efficiently implement mul-
ticast in the data plane. The proposed systems do not calculate the multicast trees, and assume that
these trees are given as inputs. One natural extension is to design a multicast routing system to com-
pute the trees. The system should allows ISPs to declare several goals such as traffic engineering,
network function chaining, and routing policies.

Another extension is to support efficient multicast in data center networks. Large-scale data cen-
ters have become essential for major content providers to offer their services, and thus to generate
more revenue. Unlike ISP networks that are constructed as graphs covering multiple geographical
regions, data center networks are often designed as multi-rooted tiered trees to achieve other goals.
Specifically, the design of data center networks focuses on achieving high bisection bandwidth, and
number of redundant paths and physical hosts. Moreover, different tree designs may have different
routing and congestion control algorithms. For example, the fat-tree data center network [8] can
support 27K hosts, 576 redundant paths between any two hosts, and 160K links using commod-
ity Ethernet switches. In addition, a two-level routing algorithm is used instead of the traditional
one-level IP prefix matching. Many data center applications may use multicast between hundreds of
hosts to perform their tasks. Examples of these applications include health monitoring, data replica-
tion and software updates. Current multicast forwarding systems incur both state and commu-
nication overheads. The proposed multicast forwarding systems may result in large label sizes because
they are not optimized for the (well-structured) data center networks.

This future work would leverage the structure of complex data center networks to design ef-
cient multicast forwarding systems. For example, the links belonging to a multicast tree across
different layers are independent, as well as links across switches of same layer. Thus, one possible
direction is to break each multicast tree into upstream and downstream paths. Each path (or set of
paths) in a layer can be encoded to a different label. Moreover, since links of same layer are indepen-
dent across the switches, we can divide these switches to virtual pods, and encode links belonging
to each virtual pod independently.
Bibliography


