CONTENT MANAGEMENT IN PLANET-SCALE VIDEO CDNs

by

Kianoosh Mokhtarian

A thesis submitted in conformity with the requirements for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2015 by Kianoosh Mokhtarian
Global video Content Distribution Networks (CDNs) serve a significant fraction of the entire Internet traffic through a global network of cache servers. Effective handling of the massive traffic at the edge is vital for the feasibility of these CDNs, which can otherwise incur significant monetary costs and resource overloads in the Internet. We study this problem from several angles. First, we design cache management algorithms for individual servers of these CDNs: an LRU-based baseline solution to address their unique requirements; a flexible ingress-efficient algorithm; a greedy offline cache aware of future requests to estimate the maximum possible efficiency; and an optimal offline cache (for limited scales). In addition, we study cross-server content management. We analyze how in the absence (impracticality) of cooperative caching, the knowledge of requests in other serving locations can lead to better caching decisions overall and reducing up to hundreds of Gbps of costly traffic. We call this practice cache coordination and design the proper mechanism for realizing it. Based on actual workload data from a large-scale, global video CDN, we first conduct a detailed analysis of the spatial correlation of video popularities—workload similarities—worldwide. We then analyze the effectiveness and feasibility of cache coordination and its scalability: from within a city to across countries. Furthermore, we study the problem of provisioning the right CDN server cluster to deploy in each location, a central problem for today’s continuously expanding CDNs. We optimize the proper server count, peering bandwidth, and server configuration as the disk drives and the necessary SSD and/or RAM cache layers to sustain the intensive I/O load. The optimization framework carefully captures the interaction of cache layers inside each server, the interplay between egress/disk capacity and network bandwidth, storage drives’ read/write constraints and their costs. Finally, we study the generic problem of large-scale message distribution in networked systems with minimum end-to-end latency. We develop a rich suite of algorithms for different message delivery requirements and application scenarios, with up to 60% reduction in delivery latency and support for several times larger scales compared to existing solutions.
Contents

1 Introduction ........................................... 1
   1.1 Motivation ........................................... 1
   1.2 Problem Statement ................................... 2
   1.3 Approach ............................................. 4
   1.4 Contributions ....................................... 5
   1.5 Organization ....................................... 7

2 Background ............................................. 8
   2.1 Internet Structure ................................... 8
   2.2 CDN Model ........................................... 9
       2.2.1 Content Management ......................... 10
       2.2.2 Traffic Mapping ............................... 11
       2.2.3 Content Caching Techniques ............... 12
       2.2.4 Server Model and Caching Efficiency ....... 12

3 Related Work .......................................... 14
   3.1 Classic Caching .................................... 14
   3.2 Cooperative Caching ................................ 15
   3.3 CDN Content Placement ............................. 16
   3.4 Video CDN Workload Analysis .................... 17
   3.5 Server Provisioning ................................ 18
   3.6 Minimum-Delay Multicast ........................... 19

4 Content Caching in Video CDNs ....................... 21
   4.1 Practical Challenges and Requirements ............ 22
   4.2 Video CDN Caching Problems ....................... 24
4.2.1 Ingress-vs-redirect Tradeoff Management ........................................ 24
4.2.2 Cache Efficiency Metric ............................................................. 25
4.2.3 Definition of Caching Problems ............................................... 26
4.3 xLRU Cache for Video CDNs ......................................................... 27
4.4 Cafe Cache for Video CDNs ........................................................... 28
4.5 Optimal Offline Cache ................................................................. 31
4.6 Psychic Offline Cache ................................................................. 33
4.7 Evaluation .............................................................................. 34
  4.7.1 Evaluation Setup ................................................................. 34
  4.7.2 Psychic: An Estimator of Maximum Expected Efficiency ............. 34
  4.7.3 xLRU, Cafe and Psychic Performance ..................................... 35
4.8 Summary ............................................................................... 39

5 Coordinated Cache Management ....................................................... 40
  5.1 Non-Coordinated Caching .......................................................... 41
    5.1.1 Metric of Interest: Cost Saving ........................................... 42
    5.1.2 Non-Coordinated Caching Alternatives .................................. 43
  5.2 Spatial Correlation of Video Popularities ..................................... 44
    5.2.1 Metrics and Popularity Classes .......................................... 44
    5.2.2 Popularity Correlations ..................................................... 46
    5.2.3 Implications ................................................................. 49
  5.3 Local Cache Coordination ......................................................... 50
    5.3.1 Coordination Mechanism ................................................... 50
    5.3.2 Experimental Results ....................................................... 53
  5.4 Extending Coordination Domains ............................................... 57
    5.4.1 Efficiency Gain ............................................................. 57
    5.4.2 Coordination Overhead ..................................................... 59
    5.4.3 Findings and Limitations .................................................. 60
  5.5 Summary ........................................................................... 61

6 Server Provisioning in Video CDNs ................................................... 63
  6.1 Adaptive Ingress Control for Minimizing Peak Upstream Traffic ........ 64
    6.1.1 Overview of Ingress Control ............................................. 65
    6.1.2 Adaptive Ingress Control Algorithm .................................... 66
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.3</td>
<td>Experimental Results</td>
<td>68</td>
</tr>
<tr>
<td>6.2</td>
<td>Layered Caches in a Server</td>
<td>70</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Mutual Effects of Layered Caches</td>
<td>70</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Analysis of Cache Layering</td>
<td>72</td>
</tr>
<tr>
<td>6.3</td>
<td>Optimized Server Provisioning</td>
<td>76</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Notation for the Optimization Framework</td>
<td>76</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Optimization Problem Formulation</td>
<td>77</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Optimization Solution</td>
<td>78</td>
</tr>
<tr>
<td>6.4</td>
<td>Evaluation</td>
<td>79</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Evaluation Setup</td>
<td>80</td>
</tr>
<tr>
<td>6.4.2</td>
<td>Experimental Results</td>
<td>83</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary</td>
<td>89</td>
</tr>
<tr>
<td>7</td>
<td>Minimum-Delay Message Distribution</td>
<td>90</td>
</tr>
<tr>
<td>7.1</td>
<td>System Model and Problem Statement</td>
<td>92</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Notation for Multicast Problems</td>
<td>92</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Formal Definition of Problems</td>
<td>94</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Routing Model</td>
<td>96</td>
</tr>
<tr>
<td>7.2</td>
<td>Minimum-Delay Multicast Algorithms</td>
<td>97</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Overview of Algorithms</td>
<td>97</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Detailed Multicast Algorithms</td>
<td>99</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Complexity Analysis</td>
<td>104</td>
</tr>
<tr>
<td>7.3</td>
<td>Evaluation</td>
<td>104</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Evaluation Setup</td>
<td>105</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Experimental Results</td>
<td>106</td>
</tr>
<tr>
<td>7.4</td>
<td>Summary</td>
<td>116</td>
</tr>
<tr>
<td>8</td>
<td>Conclusions</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>121</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Video data constitutes a major fraction of today’s Internet traffic. YouTube and Akamai are estimated to deliver up to 30% of the entire Internet traffic each—a volume in terabits per second (Tbps) scale [76, 7, 6]. Such traffic volumes cannot be distributed to worldwide users from one or a few datacenters no matter how large they are, given the limited resources of the Internet backbone [86]. Rather, the traffic is delivered to end users through a geographically distributed network of cache servers, called a Content Distribution Network (CDN).

Some medium-size CDNs consist of server racks in a few dozen serving locations at major Internet exchange points (IXPs) around the world, such as Limelight [4] and Amazon CloudFront [2]. Content providers can contract these CDNs to distribute their data. Servers at these strategic points allow a CDN to deliver the majority of the traffic to users through peering paths to their ISPs often just one hop away, minimizing the traffic reaching the far-away origin location of the data. Moreover, larger CDNs with higher traffic volumes further deploy many server racks inside the points of presence (PoP) of ISPs worldwide, such as Akamai [68], Google [3] and Netflix [5]. For example, Akamai reported their presence in about 1000 ISPs around the world with over 60,000 servers back in 2010 [68]—150,000 servers as of 2014 and growing [6]. These servers offer a win-win situation for the CDN, the ISP, and the user: minimized traffic hitting the CDN network, minimized traffic on the ISP’s uplink\(^1\), and enhanced quality of service.

\(^1\)CDN-developed servers provide more effective edge serving than a generic cache at the ISP which would have much more limited information about the content; in addition, it would have to deal with complications such as identifying the objects in the first place (interpreting URL parameters), encrypted SSL sessions, and so on. Also note that the CDN may need to pay a hosting premium to the ISP depending on the mutual leverages between the two sides.
Delivering the immense (and fast growing) volume of video traffic can incur substantial monetary costs for CDN operators and ISPs. Unlike CDNs delivering small-size, latency-sensitive content such as Web search and email, the main goals for CDNs delivering bulky video traffic at such scale are to avoid high traffic handling costs and overload on bottlenecks. In terms of latency, it is often just enough if the server-to-user RTT is maintained within a reasonable bound compared to the required initial buffering of the video.

The paramount goal of a CDN delivering voluminous video traffic is to handle as much user traffic as possible locally at the proper server location for each user network (IP prefix), and let only minimal traffic get past them and flow on unwanted/costly paths. This requires careful management of content across replica servers to maximize local serving.

Note that while one might presume the cheap cost of adding disks to cache servers as a perfect solution to this problem, the Zipfian pattern observed for video accesses [40, 102] shows that just a few percent of higher hit rates requires up to a multi-fold increase in disk size—even if ignoring the natural, constant growth of the data corpus and request traffic. That is, a linear increase in traffic saving requires exponential increase in disk cost; we study the cache sizing problem in detail in Chapter 6. The right schemes for content management are therefore of critical importance for a CDN: given the several-Tbps volume, caching content more efficiently across the CDN saves substantial amounts of traffic.

### 1.2 Problem Statement

Figure 1.1 illustrates the high-level layout of a typical CDN for delivering voluminous traffic. We refer to servers residing in the CDN network, e.g., located in its datacenters or peering points, as on-net servers and to the remote servers deployed inside the PoPs of ISPs as off-net.
The primary concern of such CDN is to keep the massive traffic away from costly/constrained paths in the Internet. For example, to serve the traffic of a given user network, *usually only one or a few server locations are preferred*, such as an off-net server location in the corresponding ISP or one behind a peering connection to the ISP; see the connections in Figure 1.1. Assigning the traffic of a user network to an arbitrary server location behind a costly transit path is not an option, unless if no alternative exists; see Section 2.1 for background information on Internet peering and transit. These traffic assignment criteria have critical implications for content management on the CDN which we discuss shortly. Note that such assignment criteria also automatically result in improved quality of service and user experience.

Given the limited traffic assignment options such as the above and the large scale of both the network and the data, many handy solutions for content delivery can no longer be applied. For example, cooperative caching of content among servers [67, 74, 89, 59], where a group of servers collectively serve a wider user population, is not easily possible. This is because different servers handle different user networks and cannot freely serve each other’s users or redirect users to each other based on the individual requested video. This would require, for instance, sending user traffic from one ISP to an off-net server location in other (competing) ISPs, or to on-net server locations with no peering path to the ISP. Similarly, DHT-based routing of requests to where the requested contents live [50, 38] is not applicable: only a few of the possibly thousands of server locations [68] should serve the (majority) traffic of a user network. Moreover, (logically) centralized content management across the CDN is extremely challenging [9, 25, 83, 18], if feasible at all, given the scale. For this, one needs to track a global view of the highly dynamic availability and popularity of millions of files at thousands of locations in an up-to-date database. Chapters 2 and 3 reviews these problems in more detail.

Rather, in the model we consider for large scale content delivery, first the traffic of each user network worldwide is assigned to the proper server locations based on traffic policies such as the above. Then, the replica servers of the CDN manage their cached contents and try to host the most popular pieces of content based on the user traffic they receive (§ 2.2). This is similar to the non-cooperative pull-based model in the literature [71, 70]. This enables a flexible, horizontally scalable CDN to keep up with the constantly growing demand.

The main problem of our interest in this thesis is effective content management on CDN servers. The feasibility of the content distribution service depends largely on how effectively the servers in each location can carry out this task and serve as a line of defense against the massive traffic; our aim is to have as little traffic as possible get past them. This traffic, in the case for on-net CDN servers, is either load the CDN’s backbone network (requiring provisioning) or possibly traffic on constrained and/or expensive Internet paths. In case of off-net servers, this incurs traffic on the ISP’s uplink which is
undesired for both the ISP and the CDN. Our goal is to maximize local serving at each server location. This is of great benefit to both the CDN and the ISPs—indeed also to user experience.

To this end, the servers need to maintain a dynamic collection of popular content at all times\(^2\). Unlike traditional caches, instead of bringing in and caching all requested videos, the servers may need to redirect a small fraction of requests through HTTP 302 (e.g., to an upstream, larger serving site of the CDN) for video pieces that are too unpopular to live on the disk—or for other practical reasons (§4.1). The servers also need to incur as little cache-fill traffic as possible while also redirecting minimal requests to other (less preferred) CDN servers for the user. Moreover, the servers need to be able to manage the inherent tradeoff between the two based on CDN configuration. Furthermore, the servers should coordinate with each other to boost the CDN’s caching performance while serving only their assigned user traffic. The CDN needs to manage such coordination between servers to the right clusters and the right scale. In addition, a global CDN needs to be constantly growing with the demand, e.g., the Akamai CDN has expanded nearly 3× in size only from 2010 to 2015 [68, 6]. For each new server deployment, the CDN needs to determine the right amount of serving capacity, cache space and network bandwidth to install. Last but not least, the thousands of CDN entities including servers, management components (e.g., traffic assignment, stats collection) and possibly end-users need to exchange information and notifications in real time. We detail each of the above problems in the respective section throughout this thesis.

### 1.3 Approach

We design several algorithms for efficient video content distribution\(^3\). Specifically, we develop a baseline, fast algorithm for the individual servers to address the challenges unique to a video CDN cache, as well as a more complex algorithm optimized for managing the tradeoff between a server’s ingress and redirected traffic. We also design two additional, offline algorithms (i.e., with knowledge of future requests) for a better understanding of the maximum efficiency we can expect from video CDN caches and assessment of our online algorithms. Moreover, we study coordination among server locations in terms of sharing content popularity information, e.g., what videos have been requested at each location and when. This allows better caching decisions locally at each server and reduces serving costs; cache coordination is not to be mistaken with cooperative caching discussed above (and in Section 2.2.1). We also conduct

---

\(^2\)Note that the contents on a server are typically highly dynamic. For example, a cache churn of 10–30% could be quite normal according to our experiments with real data (described shortly). That is, 100–300 Mbps of ingress traffic into the server updating its contents for maintaining 1 Gbps of egress traffic served to users.

\(^3\)Our solutions easily support the distribution of non-video files as well such as software updates and file downloads, which is a straightforward, less challenging task compared to video content distribution, e.g., see the challenges for partial-request patterns (§4.1), corpus size (§2.2.1), localized interests (§5), etc.
a global-scale analysis on the spatial correlation of video popularities around the world, the result of which guides server coordinations throughout the CDN. We use actual workload data from a large-scale, global video CDN to drive our different analyses and experiments in this regard. In addition, we design an algorithm, backed by different supporting analyses on actual data, for optimally provisioning each new server cluster deployment for a constantly growing CDN.

Finally, in a separate line of work we also study the generic problem of distributing messages between thousands of inter-connected nodes (a.k.a., an overlay network) with minimum latency. We design multiple algorithms for the respective requirements, with broad applications in distributed systems not limited to content delivery networks.

### 1.4 Contributions

The contributions of this thesis are as follows.

1. We identify several challenges and requirements for content caching on individual video CDN servers which cannot be addressed by standard caching solutions. We design multiple algorithms for this problem: (i) an LRU-based baseline to address the requirements; (ii) an efficient algorithm that carefully adjusts the server’s ingress and redirected traffic; (iii) a greedy algorithm for offline caching; (iv) a linear programming relaxation of the optimal offline caching problem (for small scales). Offline algorithms are aware of future requests and provide an estimate of the maximum caching efficiency we can expect from any online algorithm, i.e., with perfect prediction of access patterns. We evaluate all these algorithms with actual workload data from video CDN servers across different continents and show that they can increase caching efficiency by over 12% particularly for ingress-constrained servers.

2. We conduct a detailed analysis on the spatial correlation of video popularities around the world, which is the first such study to the best of our knowledge. This analysis is to identify similarities between workloads of different server locations and to investigate between which locations, and to what extent, the popularity of a video in one location can inform about its popularity in another. Apart from our coordinated cache management focus as part of this thesis, this analysis is of value on its own and can benefit the design of different content delivery solutions such as server placement/provisioning and global traffic mapping from user populations to server locations. Among our findings is that interests are strongly correlated within a country, even with lingual diversity such as Canada or India. Correlation across countries, however, can be of a broad range of values even between nearby
countries, depending primarily on language.

3. We present algorithms and a suite of detailed analyses for the coordination of cache servers in a worldwide CDN, i.e., sharing of content popularity information in order to make more efficient caching decisions locally at each server. We identify the various practical considerations and challenges, we build the proper coordination mechanisms and we study the following questions: (i) Can a small group of neighboring, non-cooperating cache servers improve their caching efficiency by just exchanging video request information, and what are the overheads and their tradeoff with the achievable gains? (ii) What is the role of the correlation of workloads in the effectiveness of server coordination? (iii) Is cache coordination between arbitrary servers simply ineffective or it can be harmful (compared to just leaving the servers non-coordinated)? We also investigate the possibility of expanding the coordination domain between servers from within a city to a country and across countries, to answer the following questions: (i) Can servers that have coordinated locally and already strengthened their video popularity data gain further benefit by coordinating in wider domains? (ii) Where does the increasing overhead of expanded server coordination negate its diminishing gain? (i.e., How to form the right coordination clusters across the CDN?) We find that cache coordination in its smallest form, only between a few off-net CDN locations in the same metropolitan area, can reduce the costly traffic getting past the servers by 3–11%: up to hundreds of Gbps of costly traffic—petabytes per day. However, coordination with unrelated servers even in a small group would worsen the caching efficiency. Extended coordination across an entire country is still effective and may further increase the traffic saving to over 20% with no scalability issue. However, coordinating cache servers across countries is usually harmful or yields a marginal saving (∼1%) negated by its overhead.

4. We develop an optimization framework for configuring new server deployments of a constantly growing CDN. The modeling and optimization is backed by various supporting analyses on actual data, including the effect of layering the caches in a server—a common technique to alleviate slow disks. Moreover, we develop a technique for reducing the cache ingress traffic into CDN servers specifically during their peak period, the period that matters most. This not only shrinks the peak traffic on the servers’ upstream link, but also minimizes disk writes under peak load to allow better use of the limited disk bandwidth for the large volume of reads. We examine the optimized server configuration to provision across a variety of scenarios and analyze its relationship with a diverse range of network costs, constraints, storage read/write bandwidths and prices.

5. We design a suite of algorithms for the generic problem of distributing delay-sensitive messages to a large, dynamic group of receivers with minimum latency, a.k.a., application-layer multicast. The
algorithm suite comprises multiple algorithms for different versions of the problem and different application environments: minimum-average and minimum-maximum delay to the receivers for application in small/large scale and static/dynamic settings. We also prove the hardness and the inapproximability of the studied problems to within any reasonable approximation ratio. We show through extensive experiments using real-world Internet delay datasets [96, 90] and different overlay models that these algorithms can reduce the delay observed by message receivers by up to 60% and the computation time for multicast routing trees by up to orders of magnitude (i.e., supporting several times larger scales) compared to existing approaches.

1.5 Organization

This thesis is organized as follows. Chapter 2 provides the necessary background for this thesis. This includes a brief review of today’s Internet structure and its interplay with voluminous content delivery as well as a concise description of the content distribution model on which our work is based. Chapter 3 reviews related work in several directions intersecting this thesis. In Chapter 4, we study the cache management problem on individual video CDN servers: a description of its specific requirements and the two online and the two offline algorithms introduced above. Cross-server content management via cache coordination is studied in Chapter 5. This includes a detailed analysis of the spatial correlation of video popularities worldwide (which drives the formation of cache coordination groups), the design of the proper coordination mechanism, the analysis of the gains and overheads of coordination, and a comprehensive study of its scalability. In Chapter 6, we study the problem of configuring the proper server cluster for each server location. We design an adaptive ingress control scheme for the servers (reducing the necessary network bandwidth and disk size to provision), we analyze the interactions of HDD/SSD/RAM cache layers inside each server, and we design a server provisioning optimization framework. In a separate line of work, in Chapter 7 we study the generic problem of large-scale message distribution with minimum latency and we design a suite of algorithms for different message delivery requirements and application scenarios. Chapter 8 presents concluding remarks and introduces a set of interesting open problems for future research.
Chapter 2

Background

This chapter provides the necessary background for the present thesis. We first review the connectivity structure of the Internet and analyze its implications for distributing voluminous content to the world. We then review our target content distribution model, its main components and the corresponding procedures.

2.1 Internet Structure

A high level diagram of the Internet connectivity structure is illustrated in Figure 2.1. Different networks (a.k.a., autonomous systems) connect to each other through either transit or peering connections. Transit service from network A to network B allows B to connect to the rest of the Internet through A. Transit is a paid service and is typically priced per Mbps per month based on 95%ile usage, sometimes with other constraints (e.g., a minimum usage). Peering, on the other hand, allows two networks to directly exchange traffic with each other: typically the traffic of their own downstream customers. At the core of the Internet, highlighted as “Tier 1 ISPs” in Figure 2.1, a handful of large ISPs peer with each other and form the Internet backbone. The original definition of peering is settlement-free, meaning that the two networks do not pay each other and instead earn their revenue from connecting their downstream customers. Though, paid peering also exists in today’s Internet, e.g., where traffic volume from one network to the other is notably higher than the reverse.

Besides Tier 1 ISPs reaching each other via peering in a transit-free network, other ISPs also peer with each other to minimize their transit traffic. ISPs that engage in peering while also purchasing IP transit from one or more Tier 1 ISPs are often referred to as Tier 2 networks. Tier 3 refers to networks that connect to the rest of the Internet only through purchasing transit, though these definitions are
Chapter 2. Background

Figure 2.1: Internet connectivity structure.

general and may vary in many cases.

To distribute content in this connectivity structure, a CDN forms an overlay network of servers to serve each user network (IP prefix) from the right server location. The right location is not equivalent to being geographically close, rather, it depends on topology, costs and constraints. The right location includes an off-net serving location inside the user network (Chapter 1) where applicable [68, 3, 5] or on-net locations behind a peering connection to the user network. Large CDNs such as Akamai\(^1\), Google\(^2\) and EdgeCast\(^3\) peer with many ISPs worldwide to minimize the costly transit traffic between CDN and ISP and also improve the quality of service to users\(^4\).

2.2 CDN Model

Our focus in this thesis is on the distribution of voluminous video content where the main concern is to minimize traffic on unwanted/costly paths, e.g., cache-fill traffic into servers or user traffic to non-preferred server locations. This is unlike (and not to be mistaken with) CDN designs optimizing latency to the millisecond for serving non-bandwidth-intensive content such as email and Web search [55, 21].

The two key tasks this CDN are traffic mapping and content management, sometimes performed

\(^1\)http://www.akamai.com/peering
\(^2\)https://peering.google.com
\(^3\)http://www.edgecast.com/network/map
\(^4\)It is also noteworthy that the peering landscape may be changing with different economic leverages between ISPs and content providers, such as a recent prolonged struggle between Comcast and Netflix (http://www.nytimes.com/2014/02/24/business/media/comcast-and-netflix-reach-a-streaming-agreement.html).
jointly for small CDNs [9, 18] but separately in our model as discussed shortly. The focus of this thesis is on content management as outlined in Section 2.2.1. We also visit the traffic mapping task in Section 2.2.2. We briefly review caching strategies in Section 2.2.3 and a high level server model in Section 2.2.4.

2.2.1 Content Management

Suppose a medium scale video CDN with some dozens of serving locations such as [4, 2] and a manageable corpus of files. In such a CDN, it may be possible (and more reasonable) to centrally manage content across servers, i.e., deciding how to host the files over the servers and update them dynamically with the demand [9, 25, 83], and accordingly assign users to them [92, 66, 88]. However, in a larger CDN with servers distributed in thousands of serving locations around the world and a growing corpus of millions to billions of video files, it is extremely challenging (if not infeasible) to centrally decide which video pieces should live on which servers based on a globally tracked view of the dynamic access patterns of videos per each server location; our cache coordination work in Chapter 5 provides a widened view of access patterns for the servers (e.g., country wide) and analyzes its gains and overheads, though this is not to be mistaken with the aforementioned centralized approach.

Appealing alternatives to the centralized approach include cooperative caching [67, 74, 89] and Distributed Hash Tables (DHT) [50, 38]. In cooperative caching, a group of servers collectively serve a wider user population by serving each other’s users or redirecting users to each other based on the individual requested video. Similarly, DHT based request routing sends each request to the right server content-wise, which can be an arbitrary server topology-wise. These techniques are not applicable in the context drawn above: they would require sending user traffic from one ISP to off-net servers in other (competing) ISPs or to on-net servers with no peering path to the ISP. The tight constraint is that different servers handle different user networks.

In our target CDN model, the task of content management is offloaded to the cache servers themselves based on the information available to them: the user traffic they each see (Chapter 4), and later, what their neighbors see (Chapter 5). The user traffic arriving at servers is assigned to them based on traffic mapping policies, costs, constraints and delay bounds, as detailed shortly. In other words, we do not conduct per-file request mapping. In this network of cache servers, instead of bringing in and caching all requested video files, a server may simply redirect a small fraction of requests (for unpopular videos) to other servers (§ 2.2.2)—HTTP 302. This way, each server will eventually host the files that are most popular among the users it serves. Compared to the complex/nonscalable methods for arranging files
on the servers [9, 25, 83] or repeatedly exchanging cache indexes [74, 75] (all to enforce that requests get the files always at their first point of landing), offloading content management to the individual servers and allowing a small fraction of requests to be redirected would simply suffice. This enables the valuable advantages of simplicity and easy scalability which are vital for a CDN to constantly keep up with the demand. The efficiency of this CDN depends primarily on how well the servers can manage their cached contents to directly serve as many of the incoming requests as possible with minimal cache-fill traffic and minimal requests redirected to other CDN servers (less preferred for that user IP).

**Content sharding.** To avoid content duplicates among servers in the same location and increase the depth of the caches, a common practice which our model also adopts is to shard the content ID space over co-located servers through a hash function. In other words, one may think of the servers in a location as one large server serving the whole content ID space. Therefore, when discussing cooperative caching [67, 74, 89] or cache coordination (Chapter 5), it is about servers at different locations; it is irrelevant among co-located servers. “Serving/server location” in this thesis refers to such group of co-located servers.

**Partial requests by clients.** Client players often download a video by issuing a series of byte-range requests to the server. This allows fine control over the player buffer. At the server, the first segments of a video often receive the highest number of hits compared to the rest [40]. The resultant partial caching requirements and model are described in Section 4.1.

### 2.2.2 Traffic Mapping

The traffic mapping subsystem in a CDN is to provide a mapping from each user network (IP prefix) worldwide to one or more server locations. For example, users of an ISP that hosts off-net servers of the CDN are always mapped to those servers (assuming enough egress capacity). Moreover, additional mappings may specify the destination of users redirected by the first-landed server location\(^5\). The redirection choice may include a higher level, larger serving site in a cache hierarchy which captures redirections (and cache-fill requests) of its downstream servers, besides possibly serving some user networks of its own. The redirection choice may also include a server location sibling to the initial location which also peers with the user network(s) that the initial location serves. In all cases, note that the redirection destination for a user network is chosen based on traffic mapping policies rather than the individual requested file, e.g., no redirection to off-net servers in other ISPs or to on-net servers behind costly/constrained paths. Finally, the cache-fill traffic of servers is also a substantial volume and is mapped to other (upstream)

---

\(^5\)Similarly, one may consider another map for the small volume of requests that is even redirected by the second-landed server and possibly further on, until reaching a predefined (small) cap where no further redirection is allowed on the request.
server locations, until hitting the origin of the data.

Note that this is only a high level overview of the mapping process. In practice, several constrains need to be respected and different metrics measured. Specifically, one needs to factor in the expected request volume from each user network, the available capacity of each serving location (with possible server/rack failures), the expected capacity of peering paths, the cost of alternative paths, and the measured RTTs w.r.t. given delay bounds, among other factors. The exact making of mapping decisions is outside the scope of this thesis; a related mapping system can be found in [88]. Nevertheless, note that the general structure of such map is rather simple: the vast majority of user networks are first mapped to the co-located off-net server location (if applicable) and otherwise to an on-net peering location; if none exists, they are mapped to a destination with lowest cost and/or least utilized path.

### 2.2.3 Content Caching Techniques

Caching techniques have been employed in a wide range of computing applications such as Web proxies, databases, operating systems and CPU caches. These caches serve as a protection layer to expedite query response time and minimize backend access. For the former goal, efficiency is usually measured as request hit rate and for the latter as byte hit rate (our case). The common operation of a cache is as follows: upon a request for an object that is not in the cache, fetch it from the backend and replace it on the least popular object currently in the cache. Some well known cache algorithms for replacing the least popular object are to evict the object that is Least-Recently Used (LRU), Least-Frequently Used (LFU), or even Most-Recently Used (MRU) [72].

Web caches have been extensively used throughout the Internet to minimize page load latency and traffic on backhaul links. LRU is known as the most popular replacement policy for Web caches for its simplicity and effectiveness given the temporal locality of Web access patterns [29, 14]. In large networks, a collection of hierarchical or distributed caches is employed [74]. Cooperative caching can increase the efficiency of these caches, either as request hit rate or byte hit rate, by allowing servers to serve each other’s requests [67, 74, 89]. On the other hand, for a video CDN cache server, there are several new challenges that cannot be addressed by standard caching solutions. These challenges are detailed in Sections 4.1.

### 2.2.4 Server Model and Caching Efficiency

The input and output traffic of a server are illustrated at a high level in Figure 2.2. The server is receiving requests for $D$ Mbps of demand traffic. Some of the requested content already exist in the
Figure 2.2: The input/output model of a CDN server.

cache, some are missing and should be cache-filled, and some are missing but they are too unpopular to be even brought in and replace an existing item, thus redirected (§2.2.2). The server is serving (egressing) $E$ Mbps of traffic, $I$ of which is through cache ingress. That is, a volume of $E - I$ is served directly from the cache. A fraction of requests are redirected which make up a volume of $R$. Notice that $D = E + R$.

We note that a simple cache hit rate value can no longer represent the efficiency of a cache server, since the cache may simply redirect every request that has missing segments, admit only request for existing content and obtain a (incorrect) hit rate of 100%. The redirection percentage alone is not a sufficient efficiency metric either: imagine cache-filling all misses thus 0% redirection.

Therefore, we extend the standard cache hit rate metric and define the efficiency of a video CDN cache server as the ratio of requested bytes that were served directly from the cache—not cache-filled or redirected. This efficiency can be measured as follows.

\[
\text{Cache efficiency} = \frac{D - I - R}{D} = \frac{E - I}{E + R}.
\]

For example, suppose a server operating at $(E, I, R) = (90, 20, 10)$ Mbps. That is, out of the $D = E + R = 100$ Mbps requested traffic, $E - I = 70$ is served directly from the cache, $I = 20$ is served after a cache-fill, and $R = 10$ is redirected. The efficiency of this server equals 70%. In Section 4.2.2, we generalize this metric based on the ingress-vs-redirect preference configured for the server. We can also define the server’s ingress-to-egress ratio as $I/E$, which is a metric we use in Chapter 6.

\footnote{The server may also act as a proxy, but only if its egress is under utilized and its ingress not constrained. Otherwise, proxying will just use up two servers’ resources rather than simply redirecting the user to the other server.}
Chapter 3

Related Work

3.1 Classic Caching

The standard caching problem has been analyzed extensively in the literature; see [72]. Belady’s 1966 algorithm for offline caching, i.e., caching with knowledge of future requests, evicts the object requested farthest in the future as the optimal cache replacement solution [23]. For online caching, LRU is known as the most popular algorithm used in Web caches, for its simplicity and effectiveness given the temporal locality of access patterns [29, 14]. We also use LRU as the underlying cache admission and cache replacement algorithm for parts of our work. LRU scores the objects by their last request timestamp and evicts the one requested farthest in the past upon each cache miss.

Variants of LRU such as Greedy Dual Size (GDS) [29] and GDS-Popularity [47] make it sensitive to factors such as variable object sizes. We deal with fixed-size chunks (§ 4.1) and we care about the byte hit rate (not request hit rate), thus the size of cache-filled/evicted data is not of such concern. Other LRU variants try to incorporate access frequency information (i.e., some LFU behavior) such as LRU-K [69] which scores objects by their $K$-th last access time and LNC-R/LNC-R-W3 [77] which scores objects by an estimated request rate out of the past $K$ requests. Our workload demonstrates a long, heavy tail in the access frequency distribution. Aside from hot content that will stay on the caches anyway, the files on the borderline of caching, which comprise the vast majority of fetches and evictions, lie on this tail and are usually accessed very few times during their lifetime in the cache—too few to provide meaningful frequency information for distinguishing popularities. Adaptive Replacement Cache (ARC) [65] distinguishes items accessed one and more than once, and tries to adaptively partition the cache space between the two. This is done through two LRU queues, one for items seen only once and
one for those seen twice or more, where the sum of the two queue sizes is bounded.

The above techniques and the like for caching on an individual server address the classic problem of cache replacement. For a video CDN cache, however, new challenges hinder the applicability of these techniques (§ 4.1); most notably, the decision is between cache replacement and redirection and being able to manage the consequent tradeoff between ingress and redirected traffic based on the server’s configuration. Nevertheless, to make one applicable, we have adopted the widely used LRU algorithm and adapted it for our requirements. The xLRU algorithm in Section 4.3 can be thought of as the application of existing work to our video CDN caching problem.

Furthermore, we also temporarily ignore/simplify some of these challenges in Section 5.1 to make existing caching algorithms such as LRU-2 and ARC applicable to the problem. Experiments show that the adapted LRU (xLRU in Section 4.3) still has a better performance while it is also simpler.

3.2 Cooperative Caching

Caching content on a network of cache servers has been considered in various forms in the literature. Cooperative Web caching techniques for hierarchical or distributed caching have been studied extensively, a comprehensive analysis of which can be found in [74]. Performance gains of Web cache cooperation are analyzed in [59, 89] using mathematical models and trace driven experiments. For the specific case of content delivery networks, different methods for caching and serving content to users have been proposed [71]. For example, the cache servers may exchange digests of the content they can serve [75] or report their (frequently updated) contents to a centralized directory [39]. The server may also be selected to serve users (partially) based on content hashes [67, 50, 38]. These techniques can optimize metrics such as the first-byte latency or the collective hit rate, but they are not suitable for our target CDN model. Given the scale and the limited choices to assign the requests of each user network to (§ 2.2.1), in our target model the task of cache management is offloaded to the servers themselves based on the request traffic they are each assigned.

More importantly, prior works on networked caching including the above have always considered cooperation of caches where a group of servers try to collectively serve a wider user population by serving content on behalf of each other or sending users to each other based on the requested file—likely a more profitable approach than just cache coordination (only exchanging popularity data), if it was applicable. However, in our target case a server has to serve only the user traffic assigned to itself based on traffic assignment policies and constraints, and the servers cannot arbitrarily proxy traffic through each other.

\footnote{Indeed there is sometimes a limited degree of freedom for mapping users around (§ 2.2.2), e.g., more than one peering}
Therefore, to optimize the CDN’s caching performance beyond per-individual-server algorithms, we opt to have the servers coordinate their popularity data to improve the serving of their own request traffic\(^2\).

### 3.3 CDN Content Placement

Content placement across a CDN has been studied in several flavors. The Volley system for automatic placement of static/dynamic data receives logs of user requests for the data and continuously analyzes how (interdependent) data pieces should be migrated between datacenters [9]. Borst et al. design a content placement algorithm for maximizing user traffic getting cache hit [25]. They present a linear programming formulation of the global placement problem and a distributed algorithm for cooperative caching within a cluster. The former requires global knowledge of per-server-location content popularity and availability, and the latter assigns users to arbitrary (mapping unfriendly) servers based on where content lives; see the discussion above on cache cooperation (§3.2). Tang et al. design an algorithm for placing objects on a cache network with given storage and update costs and QoS requirements [83]. The algorithm is to be run per each object. Bakiras et al. suggest to divide the cache space of a server to two partitions, one for serving as a replica and for a cache [18]. A global replica placement algorithms manages the partitioning and the content. Applegate et al. present an optimization algorithm for storing videos on the servers such that the overall cost of video transfer, based on per-GB prices for each pair of locations, is minimized [17].

All the above approaches depend on global knowledge of per-location content popularity (e.g., expected number of requests, assuming one manages to accurately quantify it) and availability. This can easily be an infeasible database to maintain in a large CDN with highly dynamic request patterns, a constantly growing corpus of millions to billions of videos and thousands of server locations. Unlike the complex/nonscalable methods for arranging files on the servers such as the above (all to enforce that requests get the files always at their first point of landing), in our model we offload the task of content management to the servers and allow a small fraction of requests to be redirected. This way, each server will eventually host what is most popular among the users mapped to it. This model enables a horizontally scalable CDN with the constantly growing demand and allows full compliance to traffic assignment criteria (§2.2)—no need for tracking a global view or for assigning user networks to arbitrary server locations.

A number of works have investigated techniques for mapping users to CDN locations. Narayana et al. on-net location to assign an IP prefix to. However, even these limited flexibilities do not overlap in terms of the user networks covered by the different server locations, not allowing us to freely perform cooperative caching.

\(^2\)Coordination is between servers at different locations. Note that coordination among co-located servers is irrelevant given content sharding (Sec. 5.3).
develop an optimization algorithm for jointly mapping user requests to the right datacenter and routing
the response back through one of the Internet service providers of the (multi homed) datacenter [66]. Xu
et al. target the same problem with a distributed algorithm [92]. These works assume the data is fully
replicated in all datacenters, making these works more suitable for applications dealing with small pieces
of data. Wendell et al. present a general content-independent mapping algorithm based on specified
policies [88]—an orthogonal work to ours which could additionally be leveraged in the CDN model we
consider.

3.4 Video CDN Workload Analysis

Video CDN workloads have been measured and analyzed by a number of earlier works including the
analysis of video popularity patterns. A long-tail Zipfian distribution for video accesses has been reported
in [40, 102], though only for a local workload at an edge network. We have observed the same distribution
in both global and local workloads which we do not reiterate in this thesis. The authors of [32, 30] examine
global popularities but report a different tail pattern: Zipfian head but no long tail. This is because the
global popularity data used in these works is based on a crawl (e.g., through “related videos”) which
gives a smaller chance to unpopular videos to appear in the data. Our collected data is uniform across all
videos in both local and global scale (Chapter 5). Chatzopoulou et al. in [31] analyze other popularity
metrics than view count such as the number of times a video is marked favorite, the comments and
ratings for different categories of YouTube videos. Zhou et al. analyze the source of video popularities
and find that the “related videos” recommendations are the main source of views for the majority of
videos on YouTube [100]. Figueiredo et al. analyze the evolution of the popularity of videos over their
lifetime [37]. Such studies provide valuable insights on video traffic aside from our (orthogonal) analysis
in Section 5.2.

As for the correlation of popularities, the authors of [40, 102, 103] find that video popularities are
local, in that the global popularity of videos and the local popularity in a certain region exhibit no
strong correlation; a description of the reasons underlying local interests and an analysis of its evolution
over time can be found in [26]. Our analysis also confirms this finding (results not repeated). However,
an important class of information for content delivery is the similarities and differences of video request
patterns from region to region across the world. Our detailed analysis of spatial popularity correlations
in Section 5.2 is the first of its kind in the literature, to the best of our knowledge, and its applications
are not limited our targeted cache coordination (§ 5.2.3).
3.5 Server Provisioning

Almeida et al. [13] study the provisioning of video content distribution networks. The authors develop a model to capture the bandwidth cost for delivering a given streaming rate from the edge caches or the origin server. The model then jointly optimizes the bandwidths to provision and the streams to store on each cache. The model is based on a global view and also does not capture the tradeoff between disk and network cost. The authors of [57] consider this tradeoff and study the joint optimization of sizing the caches and placing the right objects on them. This is based on a global view of the CDN, the individual objects, and per-client request rates and interest distributions.

Kelly and Reeves [51] study the tradeoff between storage and bandwidth cost in both global and local view. The authors analytically find the right cache sizes for a 1-parent n-children caching model and for a simple request stream of independent reference model, i.e., the probability that an object is requested is a known constant at all times independent of previous requests. For the local view, the authors relax this idealized model that ignores the temporal locality of the workload and present an analysis that takes an arbitrary sequence of requests as input and finds the optimal cache size. The analysis relies on the simple (and reliable) method of running the cache replacement algorithm on the input, rather than theoretically estimating it. This is similar to our estimation of caching efficiency for a given workload to provide part of the input to our algorithm (Chapter 6). Similar cache sizing problems to [51] have been studied for caching of peer-to-peer traffic [94], cooperative caching [62], and caching in content-centric networks [93]. These works target the sizing of classic replacement caches and only take into account cache-fills. Moreover, they assume the server as a single-layer cache with defined behavior and ignore the layering of 2+ cache algorithms in a server (§ 6.2). They also do not consider the read/write constraints of each cache, the respective prices and the way such considerations interplay with network cost.

A number of earlier works [87, 45] explicitly study the characteristics of the miss sequence of a cache, as it is often the input to a next layer cache. These studies only consider cache replacement, not cache admission and redirections which is a necessity in our model (§ 4.1). Also, given the complexity of such theoretical modeling, these works are based on idealized assumptions, most notably the independent reference model for the request stream. We find nevertheless that in the relevant scenario to our work, the addition of a cache layer atop an existing caching system does not affect its efficiency in terms of the traffic getting past the existing cache, i.e., cache-fills and (in our case) redirections (§ 6.2). The added cache, however, takes a vast read load off the existing cache, which is why it was added in the first place. On the other hand, we find that redirections can significantly change the performance of the interrelated layers, which is a factor not considered by previous studies on classic replacement caches.


3.6 Minimum-Delay Multicast

As part of this thesis, we study the general problem of delivering messages with minimum end-to-end latency from a given source to a set of receivers connected together (a.k.a., an overlay network). The model is detailed in Chapter 7. We calculate on-demand, per-source multicast trees on arbitrary mesh overlay. Compared to arranging nodes in static overlay multicast trees [19, 20], this model provides higher flexibility in selecting paths and better resilience against dynamics of the network [33, 42, 28]. Moreover, we consider source-based multicasting in which the intermediate nodes do not need to keep any per-session state information or to perform route calculations per each message. Instead, the source node calculates the routing tree and embeds it in the message (e.g., using Bloom Filters [48, 84]), and the intermediate nodes only perform simple forwarding.

Most overlay multicast algorithms focus on minimizing the link-by-link distance to the receivers [16, 33, 60, 15, 73]. The delay incurred at overlay nodes that send out each message several times, which can be a significant factor in the actual end-to-end delay as we show, is ignored in these algorithms. There have been a number of previous works that did consider the node degree problem [64, 22, 44, 80, 99], but they only try to find a routing tree in which the degree of nodes is bounded to predefined thresholds. Ito et al. [44] analyzed several forms of the multicast tree problem, and showed that it is NP-Complete to find various forms of degree-bounded trees, such as one with minimum total distance or one with minimum distance to the farthest receiver. Heuristic algorithms for constructing degree-bounded shortest path trees [64, 22] and degree-bounded minimum-diameter spanning trees [80, 79] have been proposed. It is not clear how these degree bounds are selected in practice; for instance, a fixed bound of 10 is used in [79]. In [99] it is proposed to set the degree bounds based on the level of the node in the tree. Nevertheless, these works only aim at bounding node degrees to given thresholds. Rather than bounding degrees at such coarse grain, we capture the delay caused by node degrees together with the delay incurred at overlay links as a single delay cost and minimize it.

The only previous work considering this problem, to the best of our knowledge, is done by Brosh et al. [27] who propose an approximation and a heuristic algorithm for minimizing the maximum delay in a multicast tree (the problem with minimum-average delay is not considered). However, the proposed approximation algorithm and its bound only correspond to the special case of broadcasting a message in a complete overlay graph, whereas often in practice neither the overlay is a complete graph (i.e., every node maintains and monitors a connection with every other node) nor all messages are destined to all

---

3This involves the distribution of delay-sensitive (but not significantly bandwidth-consuming) data. Examples include notification of events, stats (used for real-time decision making), availability information, real-time access control/revocation, and VoIP conferencing, among others.
nodes. Furthermore, even for this special case the approximation factor is $O(\log n)$ and $O(\log n / \log \log n)$ for directed and undirected overlay graphs, respectively, which is a considerable amount. In fact, the heuristic algorithm proposed by the authors (with no approximation guarantee) provides lower delays than the approximation algorithm while being also more efficient in running time [27]. Nevertheless, the achieved delay and the running time of this algorithm are significantly larger than our algorithms.

We also note that an important factor determining the scalability of a multicast scheme is the underlying routing protocol. The common approach used in [27] and several other works is link-state based routing [15, 16, 48, 101, 73, 56], which allows all nodes to know the full topology of the network while suffering from high overhead and limited scalability (as we will show). Our multicast scheme, on the other hand, is based on a variant of distance-vector routing and can be up to orders of magnitude more scalable.
Chapter 4

Content Caching in Video CDNs

In this chapter, we focus on the operations of individual cache servers in a video CDN for managing their contents. These caches are critical building blocks since the operating cost for a Tbps-scale CDN depends directly on their performance: to satisfy as many of the incoming requests as possible while both incurring as little cache-fill traffic as possible and redirecting minimal requests to other CDN servers (less preferred for that user IP). Determining when to bring in new content upon cache misses or to redirect requests, keeping both cache-fill and redirected traffic low, and being able to manage the tradeoff between the two based on the server’s configuration, place novel challenges before these servers which we study in this chapter.

We first analyze these challenges and the requirements for video CDN caching (§ 4.1) and formally define the problem (§ 4.2). We first develop a baseline solution built on LRU policies for addressing the requirements (§ 4.3). We then develop an algorithm that carefully adjusts the ingress and redirected traffic of a server and increases caching efficiency by over 12% particularly for ingress-constrained servers (§ 4.4). We also formulate the offline caching problem where the cache is aware of future requests, and we solve it using a greedy algorithm as well as linear programming relaxation for small scales (§s 4.5 and 4.6). These algorithms enable sound analysis of our online caching algorithms by providing an estimate of the maximum caching efficiency we can expect from any online algorithm, i.e., with perfect prediction of access patterns. We analyze all algorithms using actual video CDN data from several servers across different continents (§ 4.7).
4.1 Practical Challenges and Requirements

Recall that given the primary objective—keep the voluminous traffic on designated paths—in order to serve a given user IP usually very few server locations are preferred. Thus, mapping users to servers based on metrics unaware of such considerations, such as through a content hash [38, 67], is not feasible. Moreover, given the scale and dynamics, it is not reasonable to make (logically) centralized content management decisions. We also do not require that availability and per-location popularity of the individual files, which have no strong correlation with the global popularity [102], are tracked in a centralized directory for mapping requests to servers. In our considered CDN, first traffic of user networks is mapped to server locations, then the servers manage their own cache contents by either cache-filling or redirecting requests upon cache misses (Chapter 2).

Historically, caching techniques have been employed in a wide range of computing applications such as Web proxies, databases, operating systems and CPU caches. The common operation of these caches is as follows: upon a request for an object not in the cache, fetch it from the backend and store it based on a cache replacement algorithm. Some well known algorithms are to evict the object that is Least-Recently Used (LRU), Least-Frequently Used (LFU), or even Most-Recently Used [72]. In the class of CDNs we study, there are new challenges that cannot be addressed by standard caching solutions. These challenges are as follows.

The necessity of redirections. Each server needs to host the most popular content for the corresponding request traffic. This normally requires redirecting requests for content too unpopular to live on the disk. In addition to polluting the disk, serving unpopular content can overload the server’s disk drives with excessive writes, given the large size of video objects and their heavily skewed distribution: a large fraction of video chunks observe only one or very few accesses during their lifetime in the cache before being replaced with other (possibly unpopular) content. They incur a disk write to egress ratio of as much as 1 which is too high for the disks. Therefore, cache-filling all cache misses sometimes overflows the disks and harm not only the write-incurring requests but also the regular read operations for other requests that are simply cache hits. We observed that for every extra write operation, the server loses 1.2–1.3 reads. Thus, beyond traditional cache replacement functionalities, a video CDN server needs to be able to redirect a fraction of requests for unpopular content for (caching efficiency and) disk protection reasons.

Ingress versus redirect tradeoff. Offloading content placement decisions to individual servers enables a flexible, horizontally scalable CDN, where a server may fetch and cache the requested file upon a cache miss, or simply redirect the request. There is an inherent tradeoff between ingress and redirected
traffic: limiting redirections will start to increase cache churn and ingress, and the other way around. It is therefore not only about the cache replacement problem anymore: it is about deciding between cache-fill and redirection for each cache-miss, keeping both quantities low, and being able to manage the tradeoff between the two—a problem that has not been addressed before in other caching domains.

**Different ingress and redirection preferences.** The internal connections across a worldwide CDN may include CDN-owned or billed links. Also, the connection of the CDN network to the rest of the Internet may be through either peering or transit connections with different traffic handling costs for ISPs and CDNs. Therefore, the CDN servers can have uplinks with diverse constraints and different abilities for cache-filling data. Furthermore, another important parameter for the willingness of a server to cache-fill is the utilization of its egress (serving) capacity: for a server at which the current cache contents suffice to serve as many of the requests as can fully utilize the egress capacity, there is no point to bring in new content upon cache misses. This is because still the same volume of requests will be served and redirected, hence wasted ingress. Last but certainly not least, sometimes a server’s ingress traffic can overload the disks and harm all requests particularly at those server locations of the CDN that have smaller disks or are handling a wider spectrum of video requests, as discussed above.

In all cases, the CDN server at different locations can have different ingress capabilities for cache-filling data. While some servers may not be constrained and be able to cache-fill normally, for some others the CDN may prefer that they redirect away a good fraction of cache misses rather than cache-filling them, specially if there are appropriate alternative locations not as constrained. That is, different operating points in the tradeoff between cache-fill and redirection percentage, while all yielding the same byte hit rate, can translate into diverse consequences depending on the server. Therefore, the individual servers’ ability and willingness to cache-fill data (i.e., the CDN’s preference at that server) is an important factor that needs to be properly taken into account in caching decisions.

**Partial-request model and diverse intra-file popularities.** Client players often download a video by issuing a series of byte-range requests to the server. The first segments of a video are usually accessed much more than the rest [40]. It is thus not efficient to cache-fill or evict video files in their entirety. To simplify the support for partial caching, a common practice which we also adopt is to divide the disk and the files into small chunks of fixed size $K$ (e.g., 2 MB). This is to eliminate the complexities and inefficiencies of continually allocating and de-allocating disk space to data segments of arbitrary sizes, and it works well particularly for video (as opposed to arbitrary Web objects) given the large size of video streams compared to a chunk. However, it is still not trivial for the cache whether to redirect or cache-fill a byte-range request where the request has some chunks present and some missing in the cache. Note that although the server may store video files partially, it needs to either fully serve or
fully redirect a requested byte range: clients can request different byte ranges at their own choice from
different servers, but they do not also partially download a single byte range from multiple servers.

4.2 Video CDN Caching Problems

Client players often download a video by issuing a series of byte-range requests. Let $R$ denote a request
arriving at the server, which may be received from a user or from another (downstream) server for a
cache-fill. The request contains video ID $R.v$ and byte range $[R.b_0, R.b_1]$ and arrives at time $R.t$. The
server may serve or redirect the request to another server via HTTP 302. Although the server may store
files partially, it needs to either fully serve or fully redirect a requested byte range, as explained. Also,
recall that we divide the disk and the video files into small chunks of fixed size $K$ bytes. That is, we deal
with units of data uniquely identified with a video ID and chunk number. The chunk range for request
$R$ is $[R.c_0, R.c_1] = \lfloor R.b_0/K \rfloor, \lceil R.b_1/K \rceil$.

4.2.1 Ingress-vs-redirect Tradeoff Management

To incorporate the factors discussed in Section 4.1, namely the tradeoff between ingress and redirect
ratios as well as the preference between the two, we define a cost $C_F$ for every cache-filled byte and $C_R$
for every redirected byte, which are normalized as $C_F + C_R = 2$ (see Eq. (4.4)). We denote their ratio by
$\alpha_{F2R} = C_F/C_R$, the main configuration parameter for our algorithms that defines the server’s operating
point in the tradeoff. Note that $C_F$ and $C_R$ are not actual monetary costs to be quantified. They are
convenience variables derived from the configured value for $\alpha_{F2R}$. This parameter ($\alpha_{F2R}$) allows the
CDN to tune the operating point of a server based on the CDN’s preference at that server location
(Figure 4.5).

Sometimes it is preferred that servers ingress less traffic while a controlled increase in redirections
is acceptable and harmless. An example of this case is servers having saturated egress. These servers
will serve the same amount of traffic and redirect the same whether cache-filling normally ($\alpha_{F2R} = 1$) or
conservatively (e.g., $\alpha_{F2R} = 2$), hence wasted ingress. Furthermore, the servers may be disk-constrained,
where too much ingress and the consequent disk writes negatively impact regular disk reads. Another
example for conservative ingress is an on-net server location whose cache-fill traffic traverses the CDN’s
backbone network possibly shared with other services, and can overload it if ingressing too much. On the
other hand, the alternative location to which it normally redirects its users (without traversing the CDN
backbone) is not as constrained—a location with a larger set of racks and disks and thus a deeper cache
and less need for ingress, or a location closer to its cache-fill origin. In all the above cases, $\alpha_{F2R} > 1$
indicates that the server should limit its ingress: fetch new content only when the new file is sufficiently more popular than the least popular file in the cache. The different operating points of a server based on the value of $\alpha_{F2R}$ are analyzed in Section 4.7.

Alternatively, at some server locations, ingress and redirection make no difference and there is no gain in favoring one over the other. The most common example of this case is the vast majority of off-net server locations: remote downstream servers located in the same network as the user, from which any alternative location to redirect to or cache-fill from has the same network cost and delay as from the user. $\alpha_{F2R} = 1$ expresses this configuration for our algorithms. Finally, $\alpha_{F2R} < 1$ indicates the uncommon case of non-constrained/cheap ingress, such as an under-utilized server with spare uplink capacity.

Note that in this work, our focus is not on detailed optimization of $\alpha_{F2R}$ values across the CDN. Rather, we focus on the key building block for a video CDN which is an optimized underlying cache algorithm that complies to the desired ingress-to-redirect configuration. Given such a cache with defined operating points, the CDN will have the means for further (global) optimization. While this is an orthogonal problem to the present work described in Chapter 8, we also note that this is not a complicated task: the common configuration for servers is $\alpha_{F2R} = 1$, while a fraction of servers are ingress-constrained as explained above and can readily benefit from $\alpha_{F2R} > 1$, e.g., a default value of 2 (see Figure 4.5). Relieving these servers through optimized reduction and adjustment of ingress and redirected traffic is one of the main motivations for our algorithms in this chapter. We further design an algorithm in Section 6.1 to expand the benefits of this tradeoff control to even non-constrained servers.

### 4.2.2 Cache Efficiency Metric

Given $C_F$ and $C_R$, the cost for serving request $R$ by fetching its full byte range equals $(R.c_1 - R.c_0 + 1) \times K \times C_F$. The cost of redirecting $R$ to be served from an alternative server equals $(R.b_1 - R.b_0 + 1) \times C_R$ for the CDN and the cost of serving it directly from the cache is 0; there is indeed the cost of space and power for handling the request, which is small compared to traffic costs and is also nearly the same for the CDN for the three decisions, thus normalized to zero in the model. Note the different use of $R.b$ and $R.c$ in the above costs since a chunk is fetched and stored in full, even if requested partially. We can model the total cost of a server as follows.

$$\text{Total cost} = \text{num ingress bytes} \times C_F + \text{num redirected bytes} \times C_R. \quad (4.1)$$

We note that a simple cache hit rate value can no longer represent the efficiency of a cache, as discussed in Section 2.2.4. For the case of equal cache-fill and redirection cost ($\alpha_{F2R} = 1$), we can...
measure the efficiency of a cache server as the ratio of requested bytes that were served directly from the cache, not cache-filled or redirected (§ 2.2.4):

\[
\text{Cache efficiency (special case)} = 1 - \frac{\text{Bytes served by cache-filling + Redirected bytes}}{\text{Total requested bytes}} \tag{4.2}
\]

For the general case with arbitrary \(\alpha_{F2R}\), we generalize the cache efficiency metric as follows based on the server’s cache hits, fills and redirections with their corresponding costs.

\[
\text{Cache efficiency} = \frac{\text{Bytes served by cache-filling}}{\text{Total requested bytes}} \times C_F - \frac{\text{Redirected bytes}}{\text{Total requested bytes}} \times C_R \tag{4.3}
\]

\[
C_F + C_R = 2. \tag{4.4}
\]

Clearly, maximizing the cache efficiency metric in Eq. (4.3) is equivalent to minimizing the total cost in Eq. (4.1). Moreover, because it is only the relative value of \(C_F\) to \(C_R\) (\(\alpha_{F2R}\)) that matters for caching decisions, we can simply normalize the two as \(C_F + C_R = 2\) in Eq. (4.4). The constant 2 comes from the case with \(\alpha_{F2R} = 1\) where cache efficiency simply equals the fraction of requested bytes that were served directly from the cache \((C_F = C_R = 1\) in Eq. (4.3)). The cache efficiency metric defined in Eq. (4.3) takes a value in \([-1, 1]\). At the end, the value of \(C_F\) and \(C_R\) used in our algorithms can be obtained from \(\alpha_{F2R}\) and Eq. (4.4) as follows.

\[
C_F = \frac{2\alpha_{F2R}}{\alpha_{F2R} + 1}; \quad C_R = \frac{2}{\alpha_{F2R} + 1}. \tag{4.5}
\]

4.2.3 Definition of Caching Problems

Our video caching problems can be defined as follows.

**Problem 1** (Online Cache). Given the sequence of past requests \(R_1, \ldots, R_{i-1}\), a cache size and the current contents of the cache, make one of the following decisions for request \(R_i\) such that cache efficiency is maximized: (1) redirect the request; (2) serve the request, cache-fill any missing chunks and determine the chunks to be evicted.

**Problem 2** (Offline Cache). Given the full sequence of requests \(R_1, \ldots, R_n\) and a disk size, make one of the decisions as in Problem 1 for each \(R_i\) such that cache efficiency over all requests is maximized.

\(^1\)While a negative cache efficiency is not intuitive, we can imagine a server that is missing all requested files and is cache-filling them all—no cache hit and no redirection. This server would be performing more poorly when ingress is costlier than redirect (a negative cache efficiency) compared to when \(C_F = C_R = 1\) (zero cache efficiency).
4.3 xLRU Cache for Video CDNs

The LRU scheme, the most popular replacement policy used in Web caches, operates based on scoring objects by their last access time. The object accessed farthest in the past is treated as the least popular one to be evicted to make room for new data. For the video caching problem, the request may not need to be served at all if the file is not popular enough to be hosted.

Therefore, a video cache based on two LRU queues can operate as follows (Figure 4.1). First, a disk cache stores partial video files as chunks with an LRU replacement policy. In order to minimize cache ingress and at the same time avoid too many redirects, it is best for the server to keep only the most popular videos (from the server’s perspective). Thus, we employ a video popularity tracker on top of the disk cache to track the popularity of each video file as its last access time—how recently a chunk of the file was requested. The popularity tracking algorithm shares similarities with the LRU-2 algorithm [69]: if there is no previous request for the file, i.e., first time seeing it, the video fails the popularity test and is redirected. The same result is returned if the age of the last request for the video is older than the age of the oldest chunk on disk, a.k.a., the cache age, as shown in Line 3 of the pseudocode—ignore $\alpha_{F2R}$ for now. Otherwise, the request is sent to the disk cache for serving, including the cache-fill of any missing chunks. Out of a series of partial requests for a video in a playback session, we do the popularity test and popularity update only on the first request, and the decision is remembered for all other requests carrying the same playback ID—a unique identifier for each playback session. Otherwise, the second and further partial requests for the video in that session will always mistakenly get admitted because the first request has updated the video’s popularity.

The disk cache and the popularity tracker can both be implemented using the same data structure that consists of a linked list maintaining access times in ascending order and a hash table that maps keys to list entries. These keys are video IDs in the popularity tracker, and video ID plus chunk number in
the disk cache. This enables an $O(1)$ lookup of access times, retrieval of cache age, removal of the oldest entries, and insertion of entries at list head. Note that insertion of a video ID with an arbitrary access time smaller than the list head is not possible. Historic data that will not be useful anymore according to the cache age is regularly cleaned up.

To enable adjustment of ingress and redirected traffic, the popularity test of the xLRU scheme operates as follows. In Figure 4.1 Line 3, the popularity of a video is modeled with an approximate Inter-Arrival Time (IAT) of the requests for it, measured as $(t_{\text{now}} - t)$. Similarly, the popularity of the least popular chunk on disk, which roughly estimates the least popular video, is modeled by $\text{CacheAge}$. If the cost of cache-fill is, for instance, twice that of redirection ($\alpha_{F2R} = 2$), we expect a video to be twice as popular as the cache age, i.e., requested with a period at most half the cache age, in order to qualify for cache-fill. Thus, the redirection criteria in xLRU compares an $\alpha_{F2R}$ factor of the video’s IAT with the cache age. If greater, the requested video is not considered popular enough for the given ingress-vs-redirect preference and should be redirected.

### 4.4 Cafe Cache for Video CDNs

We develop a new algorithm for video caching that is Chunk-Aware and Fill-Efficient, hence called Cafe cache\(^2\). In a nutshell, Cafe cache estimates the joint cost of the current request as well as the expected future ones, when deciding to serve or redirect a request. This allows better compliance to the ingress-vs-redirect configuration defined by $\alpha_{F2R}$. Moreover, Cafe tracks the popularity of a video based on its individual chunks. Thus, it takes into account the intra-file diversity of chunk access patterns, and it can estimate a popularity for some chunks not previously seen as it is necessary for scoring some requested chunk ranges. In addition, Cafe cache tracks popularity as a gradually updated inter-arrival time, which prevents unpopular videos from staying for long in the cache.

In the following, we first present the high level operation of Cafe’s request admission algorithm, followed by a description of the underlying details: inter-arrival times, their implications for ordering the videos and chunks, and the consequent data structures employed by Cafe cache.

Unlike xLRU cache where popularity tracking and request admission is done at file level while disk cache is managed separately at chunk level, these two tasks are aggregated in Cafe cache. Given request $R$, Cafe cache computes the *expected cost* for serving and for redirecting the request based on the individual chunks enclosed in $R$, and serves or redirects the request accordingly: whichever incurs a

\(^2\)The highlight of Cafe’s chunk-awareness is in its computation of redirection/serving utilities for making caching decisions. This is not to be mistaken by only the support for chunk-level caching (i.e., not bringing in whole files unnecessarily) which is done by both Cafe and xLRU.
smaller cost. Let $S$ denote the set of requested chunks $([R.c_0, R.c_1])$, $S' \subseteq S$ the set of requested chunks missing in the cache, and $S''$ the set of old chunks to be evicted should the missing chunks be brought into the cache ($|S'| = |S''|$). The serving of request $R$ incurs a twofold cost: (i) the cost of cache-filling any missing chunks ($S'$), and (ii) the expected cost of redirecting/cache-filling some requests in the future—those for the chunks being evicted ($S''$). Alternatively, if the requested chunks ($S$) are considered not popular enough and get redirected, we incur the cost of redirecting the request right now and possibly in the future. This can be formalized as follows.

$$E[\text{Cost}_{\text{serve}}(S)] = |S'| \times C_F + \sum_{x \in S''} \frac{T}{IAT_x} \times \min\{C_F, C_R\} \quad (4.6)$$

$$E[\text{Cost}_{\text{redirect}}(S)] = |S| \times C_R + \sum_{x \in S'} \frac{T}{IAT_x} \times \min\{C_F, C_R\}, \quad (4.7)$$

where $IAT_x$ is the estimated inter-arrival time for chunk $x$ (described shortly), and $T$ indicates how far into the future we look. That is, we estimate the number of requests for a chunk in the near future through an imaginary window of time during which we expect the inter-arrival times to be valid—a measure of popularity dynamics and cache churn. The cache age itself, a natural choice for the length of this window ($T$), has yielded highest efficiencies in our experiments. Moreover, in Eqs. (4.6) and (4.7) we have multiplied the expected number of future requests by $\min\{C_F, C_R\}$. This is because we cannot be certain at the moment whether we will cache-fill or redirect those chunks. Most likely we will do whichever incurs a lower cost, hence the $\min$ operator. $C_F$ and $C_R$ in Eqs. (4.6) and (4.7) are derived from Eq. (4.5).

Inter-arrival times in Cafe cache are tracked as exponentially weighted moving average (EWMA) values. This enables Cafe to have IATs responsive to the dynamics of access patterns yet resistant to transient access changes. For each chunk $x$, the server tracks two quantities: the previous IAT value, $dt_x$, and the last access time, $t_x$. On a new request for $x$ at time $t$, these values are updated as follows where $\gamma (0 < \gamma \leq 1)$ is the EWMA parameter.

$$dt_x \leftarrow \gamma (t - t_x) + (1 - \gamma) \; dt_x$$

$$t_x \leftarrow t$$

Then, the IAT of $x$ at any time $t'$ can be obtained as:

$$IAT_x(t') = \gamma (t' - t_x) + (1 - \gamma) \; dt_x. \quad (4.8)$$
Cafe cache needs to maintain the chunks in a data structure ordered by IAT values, similarly to xLRU cache, although with an added flexibility which is discussed shortly. In this data structure, a chunk gradually moves down and approaches eviction unless a new request arrives and moves it up. Upon such request at time $t$, the chunk is (re-)inserted in the data structure with timestamp $ts_x(t)$. In xLRU cache, a value of $ts_x(t) = t_x$ where $t_x$ is the last insertion time (i.e., last access time) for $x$ would simply satisfy the ordering requirement: chunk $x$ is placed lower than chunk $y$ and will be evicted earlier iff $IAT_x(t) > IAT_y(t)$ at any arbitrary evaluation time $t$. Recall that in xLRU, $IAT_x(t) = t - t_x$.

This no longer holds in Cafe cache, since $IAT_x(t)$ and $IAT_y(t)$ do not solely depend on the last insertion times $t_x$ and $t_y$; they also depend on earlier insertions of the chunk reflected in their historic IAT value $dt_x$ and $dt_y$. Therefore, to (re-)insert a chunk in the right place and maintain the right ordering between chunks, we use a virtual timestamp defined as follows for (re-)insertion at time $t$.

$$ts_x(t) = t - IAT_x(t) = t - \gamma (t - t_x) - (1 - \gamma) dt_x.$$  \hspace{1cm} (4.9)

Note that unlike in xLRU where $ts_x(t) = t_x$ is a value invariant of time $(t)$, it is not in Cafe. This means that if $ts_x(t) < ts_y(t)$ when evaluated at some time $t$ (e.g., at insertion time), we need to make sure the same order holds between $ts_x(t)$ and $ts_y(t)$ when evaluated using Eq. (4.9) at a later time $t'$. This is guaranteed through the following theorem.

**Theorem 1.** By assigning the virtual timestamp of items $x$ and $y$ at insertion time with any arbitrary, fixed (imaginary) insertion time $T_0 \ (T_0 > 0)$ as $ts_x(T_0) = T_0 - IAT_x(T_0)$ and $ts_y(T_0) = T_0 - IAT_y(T_0)$, for all time $t \ (t > 0)$ we have $ts_x(t) < ts_y(t)$ iff $ts_x(T_0) < ts_y(T_0)$; indeed this only holds as long as items $x$ and $y$ are not touched again.

**Proof.** The proof is simple and based on the linear form of Eq. (4.9).

$$ts_x(T_0) < ts_y(T_0) \iff T_0 - \gamma (T_0 - t_x) - (1 - \gamma) dt_x < T_0 - \gamma (T_0 - t_y) - (1 - \gamma) dt_y$$

$$\iff T_0 (1 - \gamma) + \gamma t_x - (1 - \gamma) dt_x < T_0 (1 - \gamma) + \gamma t_y - (1 - \gamma) dt_y$$

$$\iff t (1 - \gamma) + \gamma t_x - (1 - \gamma) dt_x < t (1 - \gamma) + \gamma t_y - (1 - \gamma) dt_y$$

$$\iff ts_x(t) < ts_y(t).$$

Based on these timestamps which can order chunks by their popularities—their EWMA-ed IAT

---

$^3$For the first insertion of chunk $x$ that is never seen before, as the historic inter-arrival value $dt_x$ we use $CacheAge + 1$; we do not use $dt_x = \infty$. 

values—Cafe maintains chunks in a data structure that enables the following operations: insert a chunk with the discussed virtual timestamp \(t_{x}(T_0)\); look up the IAT of a chunk; and retrieve/remove the least popular chunks. Note that in Cafe cache, the chunks are not always inserted with a timestamp higher than all existing timestamps, unlike the case for xLRU cache where \(t_{x}(t) = t_{x}\). Rather, a chunk gradually moves up this set according to its EWMA-ed IAT value. Therefore, as a data structure that enables such insertions, we employ a binary tree maintaining the chunks in ascending order of their timestamps, as well as a hash table to enable fast lookup of IAT values by chunk ID. In other words, we replace the linked list in xLRU with a binary tree set. This enables the intended flexibility in insertions, with an insertion/deletion time of \(O(\log N)\) and retrieval of the least popular chunks \(S''\) in \(O(1)\).

Finally, we have devised a further optimization for the efficiency of Cafe cache: we would like to have an IAT estimate for chunks that are never seen before but belong to a video from which some chunks exist in the cache. Thus, we separately maintain the set of chunks cached from each video, indexed by video ID. The IAT of an unvisited chunk from video file \(v\) is estimated as the largest recorded IAT among the existing chunks of \(v\).

### 4.5 Optimal Offline Cache

Even with perfect popularity prediction and caching, the efficiency of a cache in a dynamic video CDN can be enhanced only up to a certain point. Having an estimate of this maximum is critical for understanding and improving caching algorithms. This estimates how much of the inefficiency to blame on the caching algorithms and how much on the nature of the data. We design offline caching algorithms that assume the knowledge of the complete sequence of requests.

We first formalize this problem as an Integer Programming (IP) problem and try to find the maximum possible cache efficiency via Linear Programming (LP) relaxation of the IP problem at limited scales. Then, in the next section we design a greedy algorithm that is particularly efficient in speed and memory for the actual scale of the data. While a computationally complex solution applicable to limited scales, the LP-based algorithm provides insights on where our greedy algorithm stands.

Let \(t (1 \leq t \leq T)\) denote discretized time such that \(t\) refers to when the \(t\)-th request of the sequence \((R_t)\) arrives, and \(T\) the size of the request sequence. Also let \(J\) denote the total number of unique video chunks present in the sequence and \(j (1 \leq j \leq J)\) the \(j\)-th unique \{video ID, chunk number\}. The requests can be represented with a \(J \times T\) matrix \(\{m_{j,t}\} (m_{j,t} \in \{0, 1\})\) where \(m_{j,t} = 1\) iff request \(R_t\) includes the \(j\)-th unique chunk. Similarly, we define the binary matrix \(\{x_{j,t}\}\) to hold the result: \(x_{j,t} = 1\) iff the \(j\)-th unique chunk should be in the cache at the time of handling \(R_t\). A secondary result variable
is defined as $a_t \in \{0, 1\}$ ($1 \leq t \leq T$) which indicates whether $R_t$ should be served ($a_t = 1$) or redirected ($a_t = 0$). The problem can be defined as follows.

\[
\begin{align*}
\min & \sum_{j=1}^{J} \sum_{t=1}^{T} |x_{j,t} - x_{j,t-1}|/2 \times C_F + \sum_{t=1}^{T} (1 - a_t) \times C_R \times |R_t|_c \\
\text{s.t.} & \quad x_{j,t} \in \{0, 1\} \quad (\forall j, t) \quad (4.10a) \\
& \quad a_t \in \{0, 1\} \quad (\forall t) \quad (4.10b) \\
& \quad x_{j,t} \geq a_t \quad (\forall j, t \text{ s.t. } m_{j,t} = 1) \quad (4.10c) \\
& \quad x_{j,t} \leq x_{j,t-1} \quad (\forall j, t \text{ s.t. } m_{j,t} = 0) \quad (4.10d) \\
& \quad \sum_{j=1}^{J} x_{j,t} \leq D_c \quad (\forall t), \quad (4.10e)
\end{align*}
\]

where $x_{j,0}$ is defined as 0, $|R_t|_c$ denotes the size of request $R$ in number of chunks, and $D_c$ is the total disk size in chunks. Eq. (4.10a) states the total cost which is to be minimized. The number of chunk fills is counted as $|x_{j,t} - x_{j,t-1}|/2$ since each fill comes with an eviction, both triggering a 1 in $|x_{j,t} - x_{j,t-1}|$; we can safely assume the cache is initially filled with garbage. Constraint (4.10d) ensures that if a request is admitted ($a_t = 1$), all its chunks are present or are brought into the cache. Constraint (4.10e) ensures no useless cache-fill. Although this will be taken care of by the objective function, constraint (4.10e) helps speeding up the computations.

To get a pure linear formulation that can be fed to software libraries, we introduce new variables $y_{j,t} = |x_{j,t} - x_{j,t-1}|$ and rewrite the objective function (4.10a) as follows.

\[
\begin{align*}
\min & \sum_{j=1}^{J} \sum_{t=1}^{T} y_{j,t}/2 \times C_F + \sum_{t=1}^{T} (1 - a_t) \times C_R \times |R_t|_c. \quad (4.11)
\end{align*}
\]

Moreover, the following new constraints are introduced that ensure the consistency of $y_{j,t}$. Note that the last constraint (4.12c) is only to speed up the computations.

\[
\begin{align*}
y_{j,t} & \geq x_{k,t} - x_{k,t-1} \quad (4.12a) \\
y_{j,t} & \geq x_{k,t-1} - x_{k,t} \quad (4.12b) \\
y_{j,t} & \leq 1. \quad (4.12c)
\end{align*}
\]

Although there exist libraries for (heuristically) optimizing this IP problem, our primary goal with the above formulation is finding a guaranteed, theoretical lower bound on the achievable cost—equivalently,
an upper bound on cache efficiency. We therefore solve an LP-relaxed version of the optimal caching problem by loosening constraints (4.10b) and (4.10c) to allow non-integer values in $[0, 1]$. This provides a further lower bound on cost, below which is not possibly reachable by any caching algorithm. We empirically assess the tightness of this bound by comparing Optimal cache and Psychic cache in Section 4.7. Further theoretical analysis is an interesting problem left as future work in a more theory oriented study.

### 4.6 Psychic Offline Cache

Given the large size of the data we use in our experiments, we need an offline cache with computation and memory requirements independent of the data scale. We design Psychic cache which is particularly efficient in speed and memory. Psychic cache handles a request by looking into the next $N$ requests for every chunk and it can operate as fast as our online caches, xLRU and Cafe, on the long sequence of requests.

Psychic cache does not track any past requests for the chunks. It maintains a list $L_x$ of timestamps for each chunk $x$ indicating its future requests. Also, $|L_x|$ is bounded by a given $N$ for efficiency, where $N = 10$ has proven sufficient in our experiments—no measurable gain with higher values.

Given request $R$, Psychic cache computes the expected cost of serving or redirecting the request similarly to Cafe cache, except for estimating the cost of potential future redirections and cache-fills. Instead of using an inter-arrival time computed from past requests for a chunk (Eqs. (4.6) and (4.7)), Psychic cache computes this value directly from the future requests themselves: it captures each request coming at time $t$ through an inter-arrival value of $1/(t - t_{now})$, which results in a readily computable combination of how far in the future and how frequent the chunk is requested.

$$E[\text{Cost}_{\text{serve}}(S)] = |S'| \times C_F + \sum_{x \in S'} \sum_{t \in L_x} \frac{T}{t - t_{now}} \times \min\{C_F, C_R\}$$

(4.13)

$$E[\text{Cost}_{\text{redirect}}(S)] = |S| \times C_R + \sum_{x \in S'} \sum_{t \in L_x} \frac{T}{t - t_{now}} \times \min\{C_F, C_R\},$$

(4.14)

where $S$ denotes the set of requested chunks, $S' \subseteq S$ the missing ones, and $S''$ ($|S''| = |S'|$) the chunks to be potentially evicted—those requested farthest in the future. Similar to Cafe cache, $\min\{C_F, C_R\}$ is considered as the cost of a future redirection/fill (Eqs. (4.6) and (4.7)). Also similarly, the value of $T$, which represents for how long we rely on these estimates, is set to the cache age. Note that cache age is computed differently in Psychic cache since there is no history of past requests. It is tracked separately as the average time that the evicted chunks have stayed in the cache.
4.7 Evaluation

First, we evaluate the efficiency of Psychic cache compared to Optimal cache at a limited scale. We then analyze the performance of xLRU, Cafe and Psychic caches for different setups, which include different fill-vs-redirect configurations and different server disk sizes. We finally evaluate the gains of the adaptive ingress control scheme. As elaborated in Chapter 3, we are not aware of any previous work applicable to our problem that we can include in our experiments; although, xLRU can be thought of as the adaptation of a previous work—the most widely used scheme of LRU—to our problem.

4.7.1 Evaluation Setup

Our experiments are based on actual server traces from a global CDN serving user-generated video content. The data includes anonymized request logs of six selected servers around the world: one in Africa, Asia, Australia, Europe, and North and South America, in order to cover a wide range of input profiles. These logs belong to a one month period in 2013. We replay the logs of each server to the different algorithms and measure the ingress and redirected traffic and the overall cache efficiency. When reporting average values for an experiment, the average over the second half of the month is taken to exclude the initial cache warmup and ensure steadiness.

4.7.2 Psychic: An Estimator of Maximum Expected Efficiency

Recall that Psychic cache which looks into future requests is developed to estimate how well a cache would do assuming perfect prediction of access patterns, i.e., video popularities and temporal trends. Moreover, Optimal cache incorporates the entire request sequence in an Integer Programming (IP) formulation in which the achieved cost represents a theoretical minimum for any possible caching algorithm. This minimum is further lower-bounded through LP relaxation of the IP problem.

Due to memory and computational intensity of Optimal cache, our experiments with this cache are conducted on a limited sample of the data. While this limited experiment is not as conclusive as our comprehensive experiments with the other caches, it is primarily to provide an idea of where the heuristic Psychic cache stands as an indicator of the maximum expected efficiency in the next experiments. The data for this experiment is limited as follows. We use the traces of a two day period, which we down-sample to contain the requests for a representative subset of 100 distinct files—selected uniformly from the list of files sorted by their hit count during the two days. We also cap the file size to 20 MB for this experiment. We select the disk size such that it can store 5% of all requested chunks in the down-sampled data.
We run Optimal and Psychic caches on this data and measure their efficiency. We do not include xLRU and Cafe caches in this experiment as they operate according to a history of past requests, thus unable to produce reliable results in only a two day period; they are evaluated in detail shortly. Psychic and Optimal cache, on the other hand, do not require any history and their first-hour outcome is as good as the rest.

Figure 4.2(a) illustrates the efficiency of Psychic cache compared to the LP-relaxed upper bound obtained by Optimal cache. We also plot the average, minimum and maximum delta efficiency between Psychic and Optimal caches across all six servers through the error bars in Figure 4.2(b). This figure shows that the cache efficiency achieved by Psychic is on average within 5–6% of the LP-relaxed bound. Note that an exact optimal solution, i.e., the actual upper bound on cache efficiency, is also within a gap of this theoretical bound as this bound is obtained through LP relaxation, a nonzero gap as we have observed; theoretical analysis of the tightness of this gap is left for a future study. In the remainder of this section, we use Psychic cache, our best known offline algorithm for actual scale, as an indicator of the highest cache efficiency expected from other (history-based) algorithms—one that is equally efficient in speed and memory but with Psychic prediction of future accesses even for files never seen before.

### 4.7.3 xLRU, Cafe and Psychic Performance

First, we take a close look in Figure 4.3 at the instantaneous performance of the caches: the redirection ratio, the ingress to egress percentage (i.e., the fraction of served traffic that incurred cache-fill), and the overall cache efficiency as defined in Section 4.2.2. This figure corresponds to our selected server in Europe, given a disk size of 1 TB and $\alpha_{F2R} = 2$. Also, chunk size is 2 MB and $\gamma = 0.25$ (Eq. (4.8))
Figure 4.3: Ingress, redirection, and overall cache efficiency over the 1-month period. Best viewed in color.

In this and other experiments, we can observe a diurnal pattern in both ingress and redirection for all caches, with their peak values occurring at busy hours. Overall, while the three caches incur comparable redirection rates with Cafe Cache being slightly higher, there is a significant drop of the incoming cache-fill traffic from xLRU to Cafe and Psychic, even though xLRU tries to admit only videos that are sufficiently more popular than the current contents. In other words, Psychic and Cafe caches perform more accurately in approving cache-fill for the right content. Thus, Cafe Cache achieves an average 10.1% increase in cache efficiency compared to xLRU, and the offline algorithm in Psychic achieves a 12.7% increase—respectively 0.101 and 0.127 in Eq. (4.3).

Next, we analyze the efficiency of caches for different $\alpha_{F2R}$ configurations. Figure 4.4 plots the average efficiency of the caches for this experiment on the European server given 1 TB of disk. According to the figure, when ingress is not so costly, the performance of Cafe and xLRU Caches are comparable. For example, for $\alpha_{F2R} = 1$, Cafe achieves a cache efficiency of 61% which is about 2% higher than xLRU. On the contrary, when the server is constrained for ingress, the efficiency of Cafe Cache approaches that of Psychic, as it can reduce the ingress equally effectively and hold a well selected set of popular videos.
files, hence incurring only a small increase in redirections as depicted earlier in Figure 4.3. For example, for $\alpha_{F2R} = 2$, Cafe Cache achieves an efficiency of 73%, close to the 75% of Psychic and 11% higher than the 62% of xLRU. From a cost-wise perspective, compared to xLRU, Cafe reduces the inefficiency (which translates into cost) from 38% to 27%, which is a relative reduction of 29%. We also observe a considerable gap between xLRU/Cafe and Psychic Cache for $\alpha_{F2R} = 0.5$, which is because xLRU and Cafe will (intentionally) not bring in a file of which no previous request is ever seen, while Psychic does, based on future requests.

Figure 4.4: Efficiency of the algorithms for different ingress-to-redirect configuration. Each group of 3 bars represents xLRU, Cafe and Psychic from left to right.

Figure 4.5: Different operating points of each algorithm in the tradeoff between cache-fill and redirection, governed by $\alpha_{F2R}$. The four operating points from left to right are obtained by setting $\alpha_{F2R}$ to 4, 2, 1 and 0.5, respectively.
To further understand the tradeoff between cache-fill and redirection and the way $\alpha_{F2R}$ can be used to define the operating point of each server in this tradeoff, we illustrate these points in Figure 4.5. This figure shows the ingress ratio for the algorithms on the horizontal axis and the redirection ratio on the vertical axis, for different $\alpha_{F2R}$ values and a disk size of 1 TB on the European server. Data points from left to right correspond to $\alpha_{F2R} = 4, 2, 1$ and 0.5. The figure shows that as ingress becomes more and more costly (going from right to left), all caches tend to hold onto the data stored on their disks, and redirect more requests instead. However, the ingress percentage can only be reduced to 8% by xLRU even for $\alpha_{F2R} = 4$, while Cafe and Psychic Caches closely comply with the given costs and shrink the ingress to only a few percent. On the other hand, for cheap ingress which is represented by the rightmost data points, both xLRU and Psychic suffer from high redirections as observed and discussed in the previous experiment.

![Figure 4.6: Efficiency of the algorithms given different disk capacities.](image)

We also evaluate the performance of the caches for different disk sizes, which is plotted in Figure 4.6 for the European server with $\alpha_{F2R} = 2$. As expected, efficiencies increase by providing more disk. However, xLRU results in an increasing inefficiency as disk size becomes limited, while Cafe maintains its small distance with the offline algorithm. This is because holding on to the right content on the disk and being accurate in approving cache-fill for new content, in which Psychic and Cafe caches are better than xLRU, is more critical when the disk is more limited (i.e., small cache age). In this experiment that is modeling an ingress-constrained server ($\alpha_{F2R} = 2$), to achieve the same efficiency, xLRU requires 2 to 3 times larger disk space than Cafe cache. In the non-ingress-constrained case ($\alpha_{F2R} = 1$), xLRU requires only up to 33% larger disk.

While the results reported so far correspond to one server, our experiments on the data from the
other five servers around the world have demonstrated closely similar patterns. Figure 4.7 plots one of these results: cache efficiency of the different algorithms on a 1 TB disk with $\alpha_{FER} = 2$. The same trend between the algorithms is observed across all servers. In this figure, the different levels of efficiency from server to server indicate different request profiles observed by these servers, i.e., request volume and diversity compared to the same 1 TB disk size given to all. For example, the selected server in Asia is serving less diverse requests compared to the South American one, hence higher efficiencies in Figure 4.7. Moreover, we notice a wider gap between xLRU and the other two algorithms for busier servers such as the one in South America, which confirms Psychic’s effectiveness in particular for heavy load, disk-constrained servers.

### 4.8 Summary

This chapter analyzed the unique challenges for cache management on individual video CDN servers and presented the multiple algorithms we designed for it: xLRU, Cafe, Psychic and Optimal caches. Detailed trace-based experimentation show that xLRU and Cafe caches achieve comparable efficiencies (Cafe up to 2% higher) for cases where cache ingress is not expensive, making the xLRU algorithm a compelling solution next to Cafe given xLRU’s greater simplicity in both design and computations. For servers with constrained ingress, however, Cafe cache can limit cache ingress nearly as effectively as Psychic cache that is aware of future requests and represents the best that any online caching algorithm can achieve. In these cases, Cafe cache performs with over 12% higher cache efficiency than xLRU and is within 2% of Psychic. Cafe cache also performs with an increasingly higher efficiency than xLRU as the disk size becomes limited compared to the incoming request profile.
Chapter 5

Coordinated Cache Management

In this chapter, we explore the idea that a group of CDN servers in different locations exchange video request logs, e.g., what videos have been requested recently and when. The knowledge of video requests in remote locations can lead to stronger popularity information and better caching decisions locally. This practice, to which we refer as cache coordination, does not interfere with the CDN’s traffic mapping criteria and allows the servers to still serve the same users mapped to them based on these criteria. This is in contrast to cooperative caching between servers for collectively serving a user population [89, 74, 59], e.g., sending users in one ISP to off-net servers in other (competing) ISPs or to on-net servers with no peering path to the ISP.

We use a global video workload dataset introduced below to drive our different analyses. We first investigate the key factor driving the possible success of cache coordination, which is the spatial correlation of video popularities across servers worldwide. Then, through extensive experimentation we analyze the (dis)advantages of cache coordination, its performance tradeoffs, its relationship with workload correlations, and the formation of coordination groups across the CDN. We analyze the underlying (non-coordinated) caching algorithm and present our target performance metrics in Section 5.1. We then analyze the correlation of video workloads across regions worldwide in Section 5.2. We identify the challenges for real-world coordinated caching, build our coordination mechanism, and examine coordination between neighboring server locations in Section 5.3. Finally, we analyze the scalability of cache coordination in Section 5.4.

Both our popularity correlation and cache coordination analyses show that interests are strongly correlated within a country even with lingual diversity. Correlation across countries can be of a broad range of values even between nearby countries, depending primarily on language. Moreover, we find that
cache coordination in its smallest form, only between a few off-net CDN locations in the same metro, can reduce the costly traffic getting past the servers by 3–11%: up to hundreds of Gbps of costly traffic—petabytes per day. On the other hand, coordination with unrelated servers even in a small group would worsen the caching efficiency. Extended coordination across an entire country is still effective and may further increase the traffic saving to over 20% with no scalability issue. However, coordinating cache servers across countries is usually harmful or yields a marginal saving (∼1%) negated by its overhead.

Datasets. We base our analyses in this chapter on two sampled datasets from a large-scale, global CDN serving user-generated video content. The datasets comprise anonymized video request logs for a one-month period in 2013. Each entry includes the following: timestamp, the requested video ID (hashed), byte range, playback ID, and the request origin broken down by country, province and city.

Dataset I includes traces of the entire CDN worldwide, sampled with a large factor for the feasibility of the analyses and the global simulations. This is done through random sampling by video ID to make sure we maintain the actual distribution of the workload over serving locations and over user networks. The sampling ratio (N-to-1) is properly taken care of and made transparent by the simulations, e.g., a 1 MB video piece is set to use N MB disk space and bandwidth. The results reported in this chapter are at original scale. The sampled data includes multiple billions of request entries for millions of distinct videos.\textsuperscript{1} While a representative dataset at scale, we have observed that this may leave too noisy data on a small basis such as a single off-net serving location in a local ISP. We therefore do not use dataset I for per-server simulations; the consequent limitations of our analyses are discussed at the end of Section 5.4. On the other hand, to conduct a number of such detailed experiments with minimum noise, i.e., the samples more accurately representing the actual scale, we have obtained dataset II for some smaller-scale experiment cases. This dataset includes sampled traces of a few of server locations as described in Section 5.3 and is collected with a much milder sampling filter. The sampled traces of a server rack in this dataset are roughly equivalent to the actual-scale traces of one server.

5.1 Non-Coordinated Caching

We base our cache coordination analysis on the xLRU algorithm in Section 4.3 for non-coordinated caching on each server, although our coordination mechanism is orthogonal to the underlying algorithm and is just similarly applicable to other algorithms including Cafe. Cafe is designed for managing the tradeoff between ingress and redirected traffic. In the generic case of no ingress-vs-redirect preference ($\alpha_{F2R} = 1$), the efficiency of the much simpler xLRU algorithm is comparably close to the Cafe algorithm.

\textsuperscript{1}The precise corpus size and serving capacity of the actual-scale CDN are not relevant to our analyses in this chapter and are not included for confidentiality reasons.
Our CDN-wide cache coordination experiments are based on this generic case with no per-server differentiation. This is to have a unified, symmetric view across the coordinating servers and factor out other aspects; for further discussion see Chapter 8. We compare xLRU to other alternatives shortly.

5.1.1 Metric of Interest: Cost Saving

Throughout our cache coordination experiments, we monitor the amount of undesired traffic that gets past each server, i.e., the cost for the CDN and ISPs. Suppose a server serving (egressing) \( E \) Mbps of traffic to users, \( I \) Mbps of which is by cache ingress, and redirecting \( R \) Mbps to be served by another location; see the illustration in Section 2.2.4. That is, \( E + R \) is the total request traffic seen by the server, \( E - I \) the traffic served directly from the cache, and \( I + R \) the traffic that gets past the cache either in the form of cache-fill or redirection. Similar to the cache efficiency metric in Section 4.2.2, one can define an efficiency metric for this server as \( (E - I)/(E + R) \), i.e., the standard cache hit ratio also incorporating redirections (Eq. (4.2)). However, in our current case with multiple servers involved in cache coordination, we are more interested in the aggregated amount of traffic getting past the servers (i.e., \( I + R \)) as it directly reflects a cost for the CDN—unlike the cache efficiency metric in Section 4.2.2 which is specifically designed for incorporating the ingress-vs-redirect configuration. Note that once again, the traffic served to users directly from the server’s cache \( (E - I) \) is almost free since the cost of space and power is small compared to that of the bulky traffic; this small cost can also be normalized out since it is more or less the same for the CDN whether serving from that server’s disk, serving by cache-fill or serving from a different server. However, requests served by cache-fill or redirection \( (I + R) \) generate undesired traffic. In case of off-net CDN servers, this incurs undesired traffic on the ISP’s uplink (same whether cache-filling or redirecting). In case of on-net locations, this incurs either traffic over the CDN’s backbone network, which requires provisioning, or traffic between a user network and a non-preferred server location possibly traversing constrained or expensive Internet paths.

We measure \( I + R \) as the traffic that directly concerns the CDN and we monitor its relative variations. We refer to this metric simply as costly traffic (Mbps), though note that the cost referred to is not always directly monetary, as explained above. Suppose CDN servers each receiving request for 1 Gbps of traffic with \( (E, I, R) = (800, 200, 200) \) Mbps, i.e., 400 Mbps traffic gets past the server. Suppose an alternative cache management scheme, such as coordination, improving the servers’ operating point to \( (810, 170, 190) \), i.e., 360 Mbps cost. In this case, the cost-incurring traffic for the CDN observes a 10% reduction.
5.1.2 Non-Coordinated Caching Alternatives

We have elaborated in Section 4.1 that the existing caching algorithms cannot address the requirements of a video CDN cache for various practical reasons. We therefore opt for xLRU-founded cache coordination. Nevertheless, for comprehensiveness we momentarily ignore these limitations and compare only the caching performance of xLRU (with $\alpha_{F2R} = 1$) with a number of existing algorithms. That is, we ignore the disk-write problem described in Section 4.1, and we also assume that a server can serve some and redirect some other chunks in a single multi-chunk request; we break multi-chunk requests into separate single-chunk requests and feed them to the algorithms (in other words, we force the clients to issue only single-chunk requests).

We compare xLRU to plain LRU (no redirection), LRU-2 [69] and ARC [65]. LRU-2 behaves similarly to xLRU in not storing an object upon its first request and just remembering its access time instead; this is implemented in the CDN context by proxying that first request. Though, LRU-2 differs from xLRU in scoring chunks based on the chunks’ second-last access time rather than the last. ARC [65] distinguish items accessed one and more than once and tries to adaptively partition the cache space between the two. LRU-2 performs its operations in $O(\log N)$ time and the other three in $O(1)$, where $N$ is the number of stored chunks.

We feed the logs of 15 server racks from dataset II to the four different schemes. Figure 5.1 plots the amount of costly traffic ($I + R$) incurred by the different algorithms for different disk sizes given to the servers. The total request traffic ($E + R$) is $\sim 80$ Gbps in this experiment. Disk sizes are adjusted to

![Graph showing traffic getting past the server for different schemes. The total request traffic is $\sim 80$ Gbps.](image)
the servers’ request profile: represented in the figure as a fraction of the size required to hold all video chunks requested in the first 24 hours of that server’s logs. The LRU and LRU-2 algorithms outperform one another based on the disk size: LRU-2 performs better for limited disk sizes and LRU for large disks. ARC can outperform the other algorithms only for large disks and is otherwise worse to a larger extent. xLRU outperforms the others by incurring up to 2 Gbps less costly traffic in this limited experiment. This means that if it was all the same about applicability, xLRU would still be the right candidate.

5.2 Spatial Correlation of Video Popularities

In this section, we aim to identify the correlation of workloads of different server locations and to investigate between which locations, and to what extent, the popularity of a video in one location may inform about its popularity in another. We provide several intuitive and non-intuitive findings regarding such correlations, backed by detailed numerical validation, which can guide the design of different real-world content delivery solutions including our targeted cache coordination as well as traffic mapping and server provisioning (§ 5.2.3).

5.2.1 Metrics and Popularity Classes

To quantify the similarities, we measure the correlation of popularities between two workloads \( X \) and \( Y \). The popularity of a video in our analysis is measured as the number of playbacks over the one month period. We plot each video as a data point in the \( XY \) plane based on its playback count in workloads \( X \) and \( Y \). An example is plotted in Figure 5.2(a) (in log scale) for the workload of two nearby server locations—two major ISPs in a metro\(^2\) in North America. Figure 5.2(c) shows the same plot for the workloads originated in two adjacent, same-language countries. Note that given the long-tailed, Zipfian distribution of view counts, these charts in linear scale would have almost all data points concentrated near the origin (figure not shown). Furthermore, we plot in Figures 5.2(b) and 5.2(d) the videos according to their ranks in the respective workload (zoomed). Unlike view count, rank is a metric uniformly distributed along the axes.

We quantify the similarity of popularity profiles using the Spearman correlation of popularities [81], i.e., the linear correlation coefficient of video ranks. As opposed to raw view counts, ranks have a uniform (unbiased) distribution suitable for correlation analysis, and they are not sensitive to the two workloads having different distributions, i.e., different Zipf skewness. The correlation coefficient between workloads

\(^2\)We refer to a metropolitan area as a metro, e.g., the metro of Los Angeles includes the city of Los Angeles, Long Beach and Anaheim among others.
Correlations for different popularity classes. In Figure 5.2 and the like for other cases, we observe a much stronger correlation for top videos while there is barely any correlation for the unpopular ones. This means the interests become more local as we go deeper in the popularity list. Moreover, for videos on the long tail of the popularity curve, a small number of views over a month is too noisy of a signal to meaningfully distinguish the videos in terms of popularity, hence weaker correlations. Therefore, we compute the correlation coefficient for different subsets of videos which we name as hot, popular, occasional and all videos. These groups correspond to the top 0.1% videos, top 1%, top 10% and the complete (100%) set. These correlation values are \{0.94, 0.90, 0.69, 0.10\} and \{0.61, 0.61, 0.56, 0.03\} for Figures 5.2(b) and Figures 5.2(d), respectively.
5.2.2 Popularity Correlations

We would like to identify which groups of workloads would have correlated popularities. We first examine the level of correlation between server locations in the same metro, which are expected to exhibit the strongest correlation compared to other scales. We then widen our focus and examine correlations across provinces and countries. We refer to both provinces and states simply as provinces.

Intra-Metro Correlations

A correlation value close to 0 indicates no similarity and a value of 1 indicates identical workloads. However, given the inherent dynamics and noise in video view counts, we can only expect up to a certain level of correlation value (not exactly 1) even between closely similar workloads. As a basis for understanding these levels, we analyze intra-metro correlations: the correlation of workloads between different subsets of users coming from the same metro. These workloads correspond to the servers located in and serving the same metro but in different ISPs (where applicable). Figures 5.2(a) and 5.2(b) illustrate a sample case for a metro in North America. For a given metro in which more than one server location exist, we measure the pairwise correlation coefficient between all locations. We run this analysis for a selected set of 10 metros across all 6 continents of the world. Figure 5.3(a) plots the result broken down by video popularity class. Intra-metro correlations in Figure 5.3(a) are the strongest to expect between any group of workloads across wider regions.

Intra-Country Correlations Between Provinces

In this and the following analyses, we partition the requests based on their origin location, not based on the server locations they landed on. The former demonstrates the natural correlation of local interests.
Table 5.1: Correlations of the top 5 provinces of Canada by population. Set 1 represents the average pairwise correlation of four English speaking provinces with each other (AB, BC, ON, MB) and set 2 the correlation of a French one (QC) with the other four. The numbers in each cell show the average, minimum and maximum value.

<table>
<thead>
<tr>
<th></th>
<th>Hot</th>
<th>Popular</th>
<th>Occasional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set 1</td>
<td>0.87 (0.82–0.90)</td>
<td>0.80 (0.75–0.83)</td>
<td>0.60 (0.52–0.65)</td>
</tr>
<tr>
<td>Set 2</td>
<td>0.67 (0.62–0.74)</td>
<td>0.63 (0.62–0.65)</td>
<td>0.53 (0.49–0.55)</td>
</tr>
</tbody>
</table>

Worldwide Correlations Across Countries

We investigate which countries exhibit correlations and examine two intuitive hypotheses which are geographical proximity and lingual relation. We compute the correlation coefficients between every pair of countries among a set of 80 selected countries around the world with large workloads. Figure 5.4 depicts these correlations by the countries’ distance for the Popular class. The same plot for Hot and Occasional classes (figure not shown) shows the same pattern as Figure 5.4 with Y-axis values spanning wider and narrower, respectively. The distance on the X axis is measured as the capitals’ great-circle distance, i.e., the distance on Earth’s surface.
Observation 1: Popularities in nearby countries can be of any correlation (0.2–0.7) and correlated countries can be of any distance (0–20,000 km). Figure 5.4 helps us examine the effect of geographic proximity. While there is a slight decreasing trend between distance and correlation, we find that many of the strongly correlated, nearby countries are in fact correlated due to lingual closeness rather than geographic proximity (analyzed next). A few examples are highlighted in Figure 5.4. Moreover, lingual closeness makes strong popularity correlations regardless of geographic distance.

Observation 2: Language dominates geographic proximity. To better separate the effect of language and geographic proximity, we analyze correlations between lingually related but non-neighboring countries on one hand and between nearby but lingually different countries on the other hand. We make this comparison for the two sample languages of English and Spanish and for a lingually diverse neighborhood in west Europe: English and Spanish are the most suitable candidates as two languages spoken in several, distant countries with sufficient video workload in our down-sampled traces, and west Europe the most relevant candidate as one neighborhood hosting several different languages and having enough workload. This comparison is shown in Table 5.2. In this table, the “Spanish/English” group consists of four non-neighboring Spanish speaking countries (Spain, Mexico, Argentina, Colombia) and four English (US, UK, Australia, New Zealand). Countries of each group are compared to each other (12 pairs). “Mixed neighborhood” consists of six countries in a neighborhood in West Europe, though all speaking different languages. In these sets, we see that distant countries speaking the same language exhibit a stronger correlation than nearby ones with different languages. Correlations among the latter is closer to the worldwide average between any countries.
5.2.3 Implications

The performance of a cache management scheme is a matter of properly handling videos on the mid to tail range of the popularity curve. The head which is hot content will automatically live on most servers using any algorithm and the far tail on none. In the meantime, this middle range (the Popular and Occasional classes) is the part that observes the most diverse range of correlation values\(^4\). This correlation is strong within a metro and diminishes as going to wider scales. Moreover, video popularities across the provinces of a country, even if having different local languages, are still strongly correlated. Popularities across countries are lower and more diverse, where the main determining factor is lingual closeness irrespective of geography. Geographic closeness, on the other hand, has only a minor role in shaping correlations. This information is particularly of interest for coordinated content management as it can forecast the effectiveness and guide the formation of coordination groups, as we analyze in the next sections.

4In our cache coordination mechanism (§ 5.3), servers exchange access information for all requested videos (the hot head is later filtered). The popularity classification provided here is based on a complete view of all videos requested in a region in a one-month period and is not as known to individual servers in real time. The observation under discussion is not about determining the exact range of videos to exchange access information for, since the communication overhead proves to be small in the first place. Rather, we note that it is generally for the caching of videos in the middle range of popularity that coordination could make a difference; and it is this range of videos showing the most diverse correlations.
5.3 Local Cache Coordination

This section presents and analyzes a cache coordination mechanism on top of the xLRU scheme. The idea underlying cache coordination is to (ideally) have each server of a coordinating group see requests arriving at all others: what videos and video chunks have been requested. This provides stronger popularity information for each server. It allows to admit and cache those videos that are more likely to be popular and evict those more likely unpopular, i.e., better cache admission and replacement decisions when handling requests; the breakdown is analyze at the end of this section.

We analyze the different technical considerations and implementation challenges for real-world cache coordination, we explore the design space, and we build the proper coordination mechanism in Section 5.3.1. We analyze the gains and tradeoffs of coordinating neighboring server locations in a metro in Section 5.3.2.

5.3.1 Coordination Mechanism

We refer to each group of coordinating servers as a coordination cluster, which could be as small as a single metro or as expansive as across countries. We analyze the right choice of coordination clusters in Section 5.4. Coordination clusters are non-overlapping, i.e., there is a transitive relation between coordinating servers. The sophistications of non-transitive coordination across overlapping clusters is unnecessary (§ 5.4).

Syncing the state data. Given the (intentional) simplicity of xLRU, coordination of a group of xLRU caches is a rather straightforward task to implement. Coordination is done by servers of a cluster exchanging their cache state data periodically—in particular, the part of the state that changed since the last period. Thus, the extra functionality added to the servers includes maintaining a list of videos accessed since the last data exchange and a list of their accessed chunks. The former is to be shared with the popularity tracker of other servers and the latter with their disk caches (§ 4.3). We analyze the size of this data and its tradeoff with the exchange period shortly.

The popularity data exchanged between servers includes $<\text{videoID, age}>$ and $<\text{chunkID, age}>$ entries. Age refers to the time since the video/chunk was last touched and is unaffected by the servers’ clocks being out of sync, unlike absolute timestamps. Upon receiving this data, a server reorders the entries in its two LRU queues if some videos/chunks appear fresher in the received data compared to their organic access time at the server. For either of the popularity tracker and the disk cache, denote the number of received entries by $M$ and the number of items in the LRU queue by $N$ (typically $M << N$). Assuming an LRU implementation with a hash table and a linked list arranged in recency order, this
Chapter 5. Coordinated Cache Management

operation takes a total of $O(M)$ time and is independent of $N$. This is because all the new entries from
the received data are inserted near the head, less than or equal to $M$ iterations into the local list. This
means the critical data structures do not need to be thread-locked for long for importing the data.

Data aggregation. The popularity data arriving at a location from the others includes plenty of
redundant data: popular videos appear in almost every report received, as the coordinating locations
have correlated interest profiles. Among the many appearances, only the one with the smallest age is
taken and the others discarded. To avoid the redundancy, coordination in a cluster is managed by a
coordination master, a role handled by one of the servers of the cluster. Note that alternatively pushing
this functionality to one dedicated entity on behalf of all coordination clusters, while a cleaner design and
simpler to maintain w.r.t. availability, negates the purpose of keeping traffic local (for local coordination).
Coordination in the cluster is carried out based on a pull model where the master periodically queries
all coordinating locations for their recent popularity data. This eliminates the asynchrony that might
happen if each coordinating server was to initiate a data push instead, and it also simplifies configuring
the coordination system. Once pulled everyone’s data, the master aggregates the data by taking the
minimum-age value for each video and chunk and pushes the aggregate back to the locations.

A global coordination configuration defines the coordination clusters and the master for each cluster;
there is no need for a master election procedure between the coordinating locations. The formation
of coordination clusters worldwide is discussed at the end of Section 5.4. The coordination master
is specified at rack/location level rather than an individual server and is ensured to be always up as
described shortly.

The (dynamic) sharding problem. A critical part in the cache coordination system is to isolate
the data sent to each server to only the content ID space relevant to it. The content ID space of a
server is a function of the number of racks ($N_s$) and the number of servers per rack ($N_r$) in that server
location. Specifically, to avoid content duplicates in the same location and increase the depth of the
caches, a common practice is to shard content IDs over the racks through a hash function ($\S$ 2.2.1), e.g.,
simply $\text{hash(id)} \mod N_s$ or a more complex scheme such as consistent hashing [49], and then similarly
over the $N_r$ servers of a rack.

The number of racks per location and servers per rack may vary from location to location. That is,
there is no one-to-one mapping between the servers of different location in terms of the covered content
ID space. An example is illustrated in Figure 5.5. Moreover, the sharding is not static: upon failure
of a server, one or all other servers of the rack take over its role, i.e., its IP addresses and content ID
space, until the server comes back up. This is normally handled within the rack, through a mechanism
such as electing in a failure-safe manner one of the servers as the rack’s health master that monitors and
Figure 5.5: Coordination between 3 locations. The numbers show the sharding of the content ID space over the servers in each location. This space is illustrated as 12 buckets in the figure (i.e., $\text{hash}(id) \mod 12$). The bottom node in location A rack 2 is the coordination master.

assigns shards to other servers and itself. This way, the rest of the CDN system only tracks whole racks, not individual servers. The failure handling information is not necessarily exposed to outside the rack. This means that the sharding of content IDs over the servers is not always known to other coordinating locations or the coordination master.

Flexible-shard subscriptions. One may address the sharding issue by multiplexing and demultiplexing the popularity data over sibling servers in a rack and the co-located racks in a location, through a failure-safe procedure for electing a local master in each location that is aware of the local sharding configuration. However, we opt for a simpler mechanism where the servers exchange data with the coordination master directly, as shown by the dashed lines in Figure 5.5. To eliminate redundant data, each server advertises the content IDs of its interest to the coordination master—piggybacked to its popularity data pulled by the master. We define the ID space of interest through the hash functions that result in a content ID landing on that server. For example, in Figure 5.5, \{2/3, {2, 4}/4\} represent the advertisement of the bottom server of rack 2 in location B: content IDs that fall in the 2nd bucket (out of 3) in the inter-rack sharding hash function and fall in either the 2nd or 4th bucket (out of 4) in the inter-server function.

Coordination master availability. To ensure the availability of the master of each cluster, we take advantage of the health monitoring procedure already running in each rack. That is, we pick one of the available racks in each coordination cluster at random and specify one of the IP addresses handled by the selected rack (e.g., always the highest) as the coordination master; note that rack availability
is information known CDN wide whereas server availability is not necessarily. This ensures that one and only one server picks up this role in the cluster even if the original server behind that IP address goes down. That is, there is no need to monitor and actively switch the master upon failures. Also, the configuration data does not need to be re-synchronized over the cluster upon every server failure and change of the master. Rather, the same code that is running on all the servers will periodically invoke the coordination component (e.g., every 10 minutes). This component proceeds with issuing a pull request to all IP addresses of the participant racks/locations only if it finds that the master IP of the cluster currently belongs to itself.

5.3.2 Experimental Results

We use the CDN traces in dataset II to analyze the gains of coordinating neighboring servers. This dataset contains the traces of servers in five metros in which multiple server locations exist (in different ISPs). These metros are located in Asia, Europe, North and South America and Australia to provide a diverse and representative set. The dataset also includes two small sets of servers in unrelated locations which is used for the last analysis in this section.

We feed the traces to the xLRU cache servers of each cluster in both non-coordinated and coordinated mode. The servers operate with their default configuration; we do not report the exact disk sizes for confidentiality. Our goal is to find out whether cache coordination yields noticeable benefit in the first place, what range its overhead varies in, and to what extent its success depends on the correlation of popularities. The metric of our interest is the saving in the amount of traffic that gets past the servers, a.k.a., costly traffic in Section 5.1. The different cache servers take from hours to a few days to warm up. We measure their input and output traffic in the second half of the simulated period where they all have reached their steady state.

Figure 5.6 summarizes the gains and overheads of intra-metro coordination. Coordination only within a metro yields 3–11% saving in cost-incurring traffic: a significant volume CDN-wide up to hundreds of Gbps on unwanted/costly paths. The cost saving depends on the workload correlations and the number of coordinating server locations (3 to 10+). In this experiment, the metro corresponding to cluster 2 is serving a more localized set of videos, hence higher correlations (quantified shortly), and has the highest number of server locations. Figure 5.6 also shows the impact of the data exchange period. On one hand, by increasing this period by $K$ times we increase the size of each popularity message, though at a ratio

---

5 One may alternatively conduct location-wide health monitoring over the servers of multiple racks, presenting a whole location as a large server to the rest of the CDN, i.e., individual rack and server availabilities are not known globally. Coordination configuration, however, is unaffected by such difference of granularity, since in the latter case the health system ensures the availability of some server behind every valid IP location-wide, allowing the system to specify the master at location level rather than rack level.
Figure 5.6: Cache coordination only within a metro: the impact of data exchange period on coordination overhead and gain in five metros around the world. The experimented periods are 10 sec, 1 min, 10 min, 30 min and 1 hour. The total request traffic of the five clusters is 93 Gbps and the traffic getting past the cache servers is $I + R = 24$ Gbps if no coordination.

smaller than $K$. Because the number of distinct videos and chunks watched in a 10 minute window is not as large as 10 times the same number in a 1 minute window. We plot in Figure 5.6(a) the size of the messages as a function of the exchange period for our five clusters, which confirms a slight sub-linear behavior. Each message behind this figure contains the popularity data aggregated and compressed by the coordination master (covering the whole content ID space) to be pushed back to all coordinating locations, hence called one-way overhead in the figure; the data uploaded by each location is also upper bounded by the numbers in Figure 5.6(a). Also, increasing the exchange period yields less frequent data transfers, hence a smaller amortized data rate as plotted in Figure 5.6(b).

On the other hand, increasing the period also means the servers catch up with each other’s popularity data with a larger delay—10, 60, 600, 1800 and 3600 secs in our experiments. Yet, as shown in
Figure 5.7: Intra-metro coordination gain versus the average pairwise correlation of the workloads.

Figure 5.6(c) such delay has only a minor impact in the effectiveness of cache coordination. For example, increasing the exchange period from 10 seconds to 10 minutes and 1 hour has reduced an average of only 0.1% and 0.4% in traffic saving, respectively.

The results in Figure 5.6 suggest that frequent exchange of popularity data is unnecessary. To also avoid the small loss (up to 0.6%) of effectiveness due to large exchange periods such as 1 hour—note that this number still translates into an amount of traffic in the order of 1 to 100 Mbps in the limited experiments above—an exchange period of 1 to 10 minutes appears as a proper balance. While the data exchange rate shown in Figure 5.6(b) is negligibly small in all cases (< 120 kbps), the messages may be sent as a whole rather than slowly over the period. A period of 1 to 10 min keeps the size of these messages small as well: one message of average size ~0.5 MB every minute or 2–4 MB every 10 minutes.

We have also analyzed the instantaneous gains and overheads during peak and off-peak hours of the day (figure not shown). The results show that the peak-hour message size is up to 1.8 times the daily average while the traffic saving in peak hours is also between 1.4 to 1.7 times the daily average. This keeps the above numbers still within the same range, e.g., <1 MB one-way overhead every minute. The costly traffic saved by this coordination is 1.6 Gbps.

We also would like to examine how important the correlation of popularities is to the effectiveness of coordinated caching. This analysis is not as straightforward, since the effectiveness is shaped by a compound of factors such as the number of coordinating locations and the workload volume of each, besides the popularity correlations. Also, the correlation is defined between the workload of only two locations. Nevertheless, in this experiment we try to isolate the effect of only the popularity correlation value by hand-picking 3 to 4 locations in every cluster such that they have comparable loads and nearly equal pairwise correlations. We conduct this experiment on four metro-based coordination clusters as well as two sample clusters (intentionally) comprising servers across different countries: one in Europe.
from countries speaking different languages and one in South America from Spanish speaking countries. All considered server locations are off-net servers and serve local traffic.

Figure 5.7 shows the result for each cluster as a data point on the plot. The results based on Popular and Occasional-class correlations are shown separately. The four highest savings in each figure belong to the four metros. The case with negative saving belongs to the European cluster. The case with 1.2% saving belongs to the South American (Spanish) cluster. Figure 5.7 shows that coordination gain is a direct factor of popularity correlation. Note that we do not derive a fixed, one-fits-all linear formula between the two, given the contending factors in the real world as discussed. Though, in general our different experiments (in different scales) show that coordination between three or more locations with >0.5 workload correlation in the Occasional class yields non-negligible gains and those with <0.4 is ineffective or counterproductive.

Cache coordination enables both better cache admission and better cache replacement. We analyze the breakdown of the gain between these two components to get a better understanding of the underlyings of cooperation gain and to examine whether we can obtain most of the cooperation gain with only one component. Therefore, in two separate experiments, we limit coordination to only the popularity manager and only the disk cache. The results are shown in Figure 5.8 for a metro from the previous experiment. The achieved savings by only popularity manager and only disk cache cooperation are respectively 7.5% and 5.3%, and the combined saving of both is 11%. Breakdowns in a similar range have been observed in other cases: cache admission gets a little over half of the full saving and cache replacement slightly less than half. Full cooperation is therefore necessary to realize the highest saving.
5.4 Extending Coordination Domains

We have observed that neighboring locations in a metro can coordinate their state and reduce their costly traffic with minimal communication overhead. In this section, we examine whether such coordination is also beneficial in wider domains, such as across a country or between countries. Such extension has a number of consequences on the expected gain. First, popularity profiles are not as strongly correlated across larger areas (§ 5.2.2). Second, the extension increases the communication overhead in both the size of the aggregated data and the number of its recipients. Third, coordination in smaller domains will already strengthen the popularity state maintained at the servers, casting doubt on how further helpful another layer of (wider) coordination with its consequent overheads could be.

We analyze scaling up the coordination domain in two levels: between provinces of a country and across countries. We conduct this analysis using dataset I and hypothetical server locations designated for the respective coordination units. For example, to study inter-province coordination within a country, we create one custom-provisioned server location in the simulation for handling the request traffic of each province of the target country. This is to isolate the effect of only inter-province coordination, i.e., assuming intra-province coordination is already done. In other words, each per-province server location can be thought of as the servers in that province at full coordination. Moreover, this practice assigns requests to servers based on the requests’ origin location rather than the server they happened to land on in the CDN. This eliminates the (non-organic) dependence of a server location’s workload on the particular CDN’s properties such as its existing resources and traffic mapping policies.

Similarly, in the analysis of coordination between countries we designate one large enough server location for each country. Each server location is provisioned with enough egress capacity for the incoming load and a range of different disk sizes. We simulate a server location, which in the real world consists of a series of servers handling different (non-overlapping) shards of the content ID space, as one large server with the aggregate disk and egress capacity of its individual servers.

5.4.1 Efficiency Gain

Figure 5.9(a) plots the saving percentage when the per-province servers placed in a country coordinate every 10 minutes, i.e., the extra saving added if we expand intra-province to inter-province (intra-country) coordination. The figure includes the results for six selected countries selected across different continents including two with lingual diversity. Different disk sizes are considered in these experiments to examine

---

6Note that in the real world, in addition to the many server locations handling traffic of bounded geographic areas, there still exist large serving sites covering a compound traffic of different areas of the world. The effectiveness of coordination between such locations depends more on the CDN configuration than on the servers’ geographic location and can be estimated by analyzing the correlation of the aggregate request profile they each receive (§ 5.3.2).
Chapter 5. Coordinated Cache Management

Figure 5.9: Extra saving added if we extend coordination domains, and its overheads. Disk size is normalized for confidentiality.

The saving when the servers are operating at different levels of strength: the larger the disk size, the smaller the costly traffic. This traffic ranges from tens to hundreds of Gbps per country in the above experiments with different disk sizes. The incremental saving in costly traffic in this expansion step ranges from 4% to 8% in most cases.

It is interesting to note that in the two countries with lingual diversity among their provinces, Canada and India, the smallest saving is achieved, yet still positive: about 3–4%. We ensure that it is not only the overall saving that is positive (e.g., there is positive gain even in coordinating the servers in Quebec with the other Canadian provinces) through another experiment which isolates pairs of provinces and confirms the gain. For instance, the saving between the provinces of British Columbia (English) and Quebec (predominantly French) is 1.5% whereas this number is 2.2% between British Columbia and Ontario (both English); these small magnitudes are due to having only two coordinators in the isolated experiment. It is noteworthy that in the case with 10X disk size, a slight anomaly is observed. In this case, a number of servers in the experiment take a long time to fill up and reach their steady state, even

(a) Added saving by extending coordination from intra-province to inter-province.

(b) Added saving by extending coordination from intra-country to across countries.

(c) The overhead for Figures 5.9(a) and 5.9(b).
longer than a month which is the duration of our dataset. This has affected the measured saving amount, which is the average value over the second half of the month, and caused a drop most significantly for India and to a smaller extent for France and Brazil. The otherwise increasing trend in cost saving with disk size indicates that by growing the depth of the caches, we deal with unpopular videos and chunks watched less and less frequently. The increasingly noisy information on a video’s popularity could better benefit from the additional information coming from other caches.

We analyze inter-country coordination in two classes: between countries in the same continent and between countries speaking the same language irrespective of their continent. We conduct the continent-wide experiment in Europe, South and North America: from significant language diversity to no diversity (for this reason Mexico is considered South America). We conduct the language-based experiment between 6 English and 19 Spanish speaking countries. The results show shortly that examining other continents and languages is unnecessary. Figure 5.9(b) illustrates the saving in each of the cases. Coordination of countries of different languages worsens their performance, hence a negative saving. This harm can be as significant as $-11\%$ for the case with Europe and $-4\%$ for South America where most countries speak Spanish but combined with Portuguese and smaller populations of French and Dutch. North American servers (US and Canada) can achieve a minor saving of up to $0.7\%$ by extending their coordination from intra-country to between the two countries. If including another 4 English speaking countries, the saving grows to $[1\%, 1.4\%]$. This saving is smaller between Spanish speaking countries and is in $[-0.5\%, 1.1\%]$, meaning a more local interest in those countries. In particular, the saving is positive only for large disks, which is because such cache spaces can hold more and more unpopular videos. The increasingly noisy popularity information for such videos benefits more from the additional information received via coordination.

5.4.2 Coordination Overhead

Figure 5.9(c) plots the overhead of exchanging data for the above experiments. The values represent the average rate and size of the data sent by the coordination master to each coordinating location (after compression). The size of the data in the reverse direction, i.e., pulled from each location, is therefore upper bounded by the plotted value too. Note that the servers’ disk size is irrelevant to the message sizes. Figure 5.9(c) shows how the messages expand by extending the coordination domain. This is due to the growing number of distinct videos and chunks watched in the past 10 minutes when extending our view from a metro to a country and continent. The size of the messages distributed country-wide every 10 minutes is in most cases up to 60 MB which translates into $<1$ Mbps of amortized overhead if
transmitted over the 10 minute period; the resultant delay in sending the last piece has no noticeable impact on the achieved saving as analyzed in Section 5.3.2. For inter-state coordination in the US, this overhead is 3.5 Mbps, i.e., a more diverse range of videos are watched in the US. Similarly, a broader range of videos are watched in English speaking countries than Spanish, though more strongly correlated across the former (Figure 5.9(b)).

Note that these numbers represent the overhead traffic from the coordination master to each server location. This overhead is typically in the order of 0.1% of the costly traffic of each designated server location in our experiments—one per province or country. However, when going to the actual individual server locations of a CDN such as all off-net locations in small ISPs, this overhead can be over 1% of the location’s costly traffic. This cancels out the saving achieved by inter-country coordination. For coordination within a country, the total overhead and consequently the exact remaining profit of coordination are CDN-specific and depend on the exact number of locations where the CDN has deployed servers. Nevertheless, our results suggest that country-wide coordination is generally an advisable practice. For example, given the 3.5 Mbps overhead and assuming a 100 Gbps saving in the costly traffic by inter-state coordination in the US, the total overhead can approach the saving (and render the coordination ineffective) if the number of server locations just in the US reaches 30,000 which is unlikely.

5.4.3 Findings and Limitations

We have found that country-wide coordination is profitable in our examined countries with small overhead (<4 Mbps per server location) and an added cost saving of 4–8% on top of any saving by province-wide and smaller scale coordinations. This includes even multi-lingual countries Canada and India. Therefore, country-wide coordination is generally a profitable practice. However, coordinating servers across countries is not advised, as even the highest gain it can achieve can be outweighed by the introduced overhead. Up to country-wide coordination is the right choice of clusters, and overlapped coordination clusters would not become necessary.

We have also found that being in the same nation is a stronger factor in having similar workloads than speaking the same language across different countries, e.g., compare the coordination gain of Canadian provinces (including Quebec) and that of English speaking countries: Figures 5.9(a) and 5.9(b). These results match the findings in Section 5.2 on workload correlation levels within and between countries.

We have analyzed the incremental traffic saving that can be achieved by extending the coordination domain separately in each step. The percentage values provide an understanding of each extension step as well as a rough estimate of the compound saving from non-coordinating caching to country-
wide coordination. However, we cannot quantify an exact value for the compound saving of the actual CDN server locations coordinating, rather than the ones designated for each province/country in our experiments. This is because our dataset is down-sampled with a small factor for practicality reasons. It currently contains multiple billions of requests for millions of distinct videos. While a representative sample on a country-wide basis (and province-wide for the experimented large countries), the small data that lands on each individual server location is not an accurate sample for realistically simulating many of the locations with no more than a handful of servers—the case with a considerable fraction of off-net locations. We also have not analyzed province-wide coordination between cities/metros. Besides having too small data on a per-city basis, our traces also lack city-breakdown information for some parts. The corresponding gain can be assumed having values between metro-wide savings (4–11%) and country-wide (3–9%). At the end, the compound gain can be estimated as the aggregate of the gain of metro-wide coordination (where applicable; 4–11%), the gain of extending metro-wide to province-wide coordination, and the gain of extending province-wide to country-wide coordination (3–9%): conservatively, from 10% to over 20% saving in the costly traffic. Exact quantification of the compound saving through modeling and/or modeling-assisted simulation is left for a separate study in Chapter 8; note that given the substantial volume of traffic, even the smallest end (<10%) yields up to hundreds of Gbps traffic (petabytes per day) cut on unwanted/costly paths.

5.5 Summary

This chapter analyzed the benefits and the scalability of distributed cache coordination across a global video CDN. First, analyses based on real CDN workloads showed that video popularities are strongly correlated within a metro or even a country, even with lingual diversity (>0.5 correlation). This translates into a positive gain in cache coordination, as confirmed by the coordination experiments. Correlations across countries are diverse irrespective of geography, where language is the main driving factor. Though, even across countries of the same language, interests may be more local (Spanish countries) or less (English). Furthermore, a cache coordination mechanism was designed for real-world CDNs and it was shown that exchanging cache state data as infrequently as every 10 minutes would simply suffice while it reduces the overhead to be negligible. Moreover, the results showed a close relationship between the correlation of workloads and the profitability of coordination. The results also showed that arbitrary coordination can actually worsen the caching performance compared to simply leaving the caches non-coordinated. Scalability analysis of cache coordination, which guides the formation of coordination groups throughout a CDN, revealed that up to country-wide server coordination is beneficial even in
countries with lingual diversity, while coordination between countries is generally ineffective or harmful given the associated overheads.
Chapter 6

Server Provisioning in Video CDNs

A content distribution network has to be constantly growing in order to keep up with the demand. For instance, the Akamai platform has expanded nearly 3× in size from 2010 to 2015 [6, 68]. The constant growth of a CDN includes adding server capacity to existing server locations, and more importantly, having presence in more and more locations to maximize serving at the edge. This includes installing on-net server racks in new datacenters or Internet exchange points (IXPs) and installing new off-net ones in ISP PoPs around the world.

Proper provisioning of the deployments is a critical task that determines the CDN’s cost and efficiency and is the problem we study in this chapter.

In an oversimplified view, this problem includes finding the right cache size to provide to achieve a desired caching efficiency at each target location. The practical problem, however, poses more fundamental challenges. First, the right caching efficiency to target is not a given, rather a central parameter to optimize. Due to the power-law distribution of video popularities [40], gaining every few percentage points of cache efficiency requires exponential increase in cache size. The consequent cost may easily outweigh that of just settling for a lower cache efficiency and letting some costly traffic get past the server in the form of cache-fill or redirection. Second, a certain number of deployed servers can egress up to a certain capacity. Over-provisioned server clusters that rarely get to full utilization are a waste of hardware and space. Smaller deployments, on the other hand, result in more costly traffic which may be reasonable to some extent or not at all, depending on the location. This traffic is the excess demand that is going to be served from another server location, e.g., an upstream, larger one. Third, cache servers need to designate RAM and/or SSD caches above the disks given the excessive amount of I/O, as we analyze. To sustain some expected egress traffic and cache efficiency, a non-trivial combination of cache

layers is required in a server to stand the read and write loads. Depending on their r/w bandwidths and prices, cache layers in a server fundamentally change the classic caching tradeoff between cache size and network bandwidth. In addition, the layered setting affects the behavior of caching algorithms; they may diverge from their expected size-vs-efficiency operating curve for the workload.

We analyze these and other considerations for deploying new CDN servers at a given location and we build the proper optimization framework. In particular, we first develop an adaptive ingress control technique, based on our flexible caching algorithms xLRU and Cafe (§ 4.3 and 4.4), to minimize the ingress traffic into the servers specifically during peak hours. The peak time traffic determines the upstream cost as well as the necessary disk bandwidth, as detailed below. The adaptive ingress control technique is presented in Section 6.1 and is of value of its own irrespective of the following server provisioning framework. Then, in Section 6.2 we conduct a detailed analysis, through actual video CDN traces, of the effects of layering the caches in a server, such as disks, SSDs and memory. We investigate whether and how the behavior of caches would change in such layered settings. This behavior is the basis of any cache provisioning study. We then develop an optimization algorithm in Section 6.3 for finding the right server count and server configuration (disk drives and the necessary SSD/RAM caches) to deploy in a location and optionally the peering bandwidth to provide. The process carefully captures the interplay between egress/disk capacity and the upstream traffic getting past the servers, the effect of layering on the caching algorithms, and the read/write constraints and the prices of cache storage drives. Finally, in Section 6.4 we analyze the optimal deployment cluster along multiple dimensions and study its relationship with network costs, constraints, storage read/write bandwidths and prices.

6.1 Adaptive Ingress Control for Minimizing Peak Upstream Traffic

We have seen in Chapter 4 that the algorithms we designed, xLRU and Cafe cache, can tune a CDN server’s operating point based on a given ingress-vs-redirect configuration parameter ($\alpha_{F2R}$). In this section, we design an adaptive ingress control scheme to further improve the efficiency of the servers. This scheme takes advantage of the flexibility of Cafe and xLRU and adjusts $\alpha_{F2R}$ at runtime based on the egress load. It minimizes the ingress traffic specifically during periods of peak load, the periods that matter most. This enables two important gains.

1. It reduces the peak-time upstream traffic getting past the servers. This is of paramount importance for both the CDN and the ISPs. It is the peak traffic on the servers’ uplink that determines the
required infrastructure or peering bandwidth to provision and/or the transit bandwidth cost which is often priced at 95%ile usage (§ 2.1). In the case for off-net server, it further shapes the ISPs’ costs and traffic gains if deploying our servers; an important competitive edge across the multiple CDNs that are now competing for presence in ISPs’ PoP space.

2. It can cap the maximum cache-fill and consequently disk-write rate. Excessive disk writes can easily overflow the disks and harm not only the write-incurring requests, but also the regular cache-hit traffic (§ 4.1). Limited disk read bandwidth can be alleviated by employing memory and SSD caches above the disks, but the write bandwidth cannot. Our adaptive ingress control technique minimizes disk writes and allows to better utilize the disk bandwidth for more and more reads specifically under peak serving load.

### 6.1.1 Overview of Ingress Control

To illustrate the underlying idea, suppose a cluster of off-net servers with egress capacity $E_{\text{max}}$ that are serving their assigned user networks with demand $D(t)$ Mbps at time $t$. These servers would usually get saturated during busy hours ($D(t \in \{\text{busy hours}\}) \geq E_{\text{max}}$), where the excess traffic along with the normal redirected and cache-fill traffic of the off-net servers are sent upstream, such as to a larger on-net serving site; see Figure 6.1. To get a numeric sense, the servers usually incur a cache-fill traffic of roughly 20% of their served traffic in our experiments, although this ratio can widely vary across servers and configurations (§ 4.7). During the peak period, many of the off-net servers’ cache-fills for new content can be considered redundant because the server will still be serving the same egress traffic (at capacity) and redirect the same. On the other hand, turning off cache-fills altogether during the peak period means redirecting away every cache miss and can easily leave the server’s egress capacity under-utilized despite the peak load, as we see. In other words, the server should try to *capture and serve a higher number of requests for content already existing in its cache*, including many requests from the excess demand that are normally sent upstream. This is a challenging task.

---

*Server racks not fully utilized even during peak hours would be a waste of hardware/space that neither the CDN prefers to deploy nor the ISP to host.*
Scale and dynamicity. The discussed control is straightforward in small scale CDNs where the individual video chunks cached on each server are known when making request mapping decisions. At the scale of CDNs we consider, we cannot expect the request mapping system to be aware of the millions of individual video pieces dynamically entering and leaving the thousands of servers at a rather high rate—instead the request traffic of user networks is assigned to server locations primarily based on traffic assignment constraints (§ 2.2). We aim at providing a loose control for the servers themselves over the requests they receive, while respecting all the constraints.

Request mapping and the server’s view. Depending on the specific mapping strategy implemented by a CDN, the servers in a location may or may not always see the entire request traffic of the corresponding user networks when \( D(t) \geq E_{\text{max}} \). In one naive way of mapping, the CDN may send the entire demand of a network to the corresponding off-net servers to simply have the excess portion overflow to other servers upstream. In this case, the off-net servers always see the entire request traffic. Alternatively, a more sensible traffic mapping approach assigns user requests to servers only up to their capacities. For example, the request mapping system can implement such a rate control for each user network and the corresponding server location by assigning only a certain fraction of the request traffic at random to the target servers, and the rest to upstream\(^3\). Alternatively, the mapping system may continuously track the volume of requests previously assigned to each location in a (sliding) small window of time and assign new requests only if they do not exceed the capacity limit during the window. In any case, this is a decision not aware of individual video chunks currently living on each server.

6.1.2 Adaptive Ingress Control Algorithm

Our ingress control scheme is based on the Cafe (or xLRU) algorithm. Cafe allows the servers to disfavor cache-fills to some give extent and instead redirect away the necessary portion of cache misses. For a server with identical ingress and redirected path, this is a trade of no gain during off-peak hours. Though, at peak times this practice can bring significant savings for the CDN and the ISP. This saving, which is not reflected in the individual servers’ cache efficiency, is as follows. This is described for a sample cluster of off-net servers, while the same principles applies to on-net server locations as well.

Suppose there exists \( D \) Mbps of demand, the off-net servers are serving \( E \simeq E_{\text{max}} \) Mbps of egress and redirecting \( R \) Mbps, and \( I \) Mbps of the egressed traffic incurs cache ingress; suffix \((t)\) is removed from the notation for brevity. That is, the servers are assigned and see requests for \((E+R)\) Mbps of request traffic. During peak hours, the user network’s possible excess demand of \( D-(E+R) \) (if \( >0 \)) is not sent to the

\(^3\)For instance, sending a fraction of \( E_{\text{max}}/D(t) \) of the request traffic to the servers ensures fully utilized but not overloaded egress. Though, to account for the requests the server may redirect, this fraction should be updated to \( E_{\text{max}}/D(t) \times 1/(1-r) \) where \( r \) is the server’s current redirection rate.
off-net servers and is served directly by some alternative (upstream) location. Thus, the total upstream traffic including the excess demand, cache-fills and redirections equals $D-(E+R)+I+R = D-E+I$; see Figure 6.1. Reducing ingress at the server to $I' < I$ and in turn increasing redirects to $R' > R$ translates into a total upstream traffic of $D-E+I'$ at peak compared to $D-E+I$, which is a gain of $(I-I')$ Mbps$^4$.

Notice that this is only true as long as we maintain $E \simeq E_{\text{max}}$ despite increased redirections. Our ingress control scheme carefully tightens the filter for approving cache-fill in order to gradually reduce $I$, which in turn increases $R$. The increased redirections result in capturing a little more requests; recall that the request traffic assigned to the server is (eventually) $E+R$. This adaptation of $I$ and $R$ is done to the proper extent to always keep the egress near $E_{\text{max}}$. Once the demand drops from its peak, the server reverts back to normal cache-fill and redirection and updates its cached contents as necessary, as the experiments illustrate shortly.

The discussed saving is not to be mistaken with and is not reflected in the individual servers’ caching efficiency metric. While the latter quantifies efficiency per server and is the metric we use to compare caching algorithms, the former saving views efficiency at a higher level picture—for the whole traffic of an ISP—given the underlying Cafe building block with optimized server-level caching efficiency.

Also note that the discussed method changes the request traffic arriving at the CDN servers upstream the off-net location: the volume of this traffic is reduced by $(I-I')$ Mbps and the request profile is changed since more requests for popular videos stop downstream. However, this does not affect the performance of the upstream servers. First, the upstream is typically a larger serving site handling the cache-fill and redirected traffic of a large group of smaller downstream sites. Second, the behavior of a cache server (in here, the upstream one), particularly in terms of its ingress and redirected traffic, either is unaffected or changes only negligibly by another smaller cache preceding it and hiding from it requests for popular content (§ 6.2).

We develop a control loop that dynamically adapts $\alpha_{F2R}$ to achieve the sketched goal. This algorithm is illustrated in Figure 6.2. Once the server detects the demand is approaching its peak and there may exist a higher volume of request traffic to be assigned to it (Line 2), it starts gradually leaning towards more redirects than ingress (Line 3). On the other hand, once the egress volume starts falling below a threshold near the maximum capacity (Line 4), which could be due to descending demand past the peak hours or due to an overly raised $\alpha_{F2R}$ bar in Line 3, the server retreats back.

An updated $\alpha_{F2R}$ may take some time to show its effect on the egress and redirected traffic ($I$ and

---

$^4$Having more requests observe a redirection is not completely neutral. It may introduce an additional delay compared to having directly landed on the destination server instead. However, this delay is marginal particularly compared to the initial buffering of a video stream. This is because the redirection step is from a server inside the same network as the user with very small round-trip delay, and the additional host name resolution for the hosted off-net server is almost always a local DNS hit at the ISP.
Adaptive ingress-vs-redirect tradeoff control

// Called every $K$ minutes

PeriodicAdaptation($K, \alpha_{F2R}, THRESH_{HIGH}, THRESH_{LOW}, E_{cap}, AGGR, MAX_{\alpha_{F2R}}, MIN_{\alpha_{F2R}}$)

1. $E = \text{Smoothen}(\text{AverageMinutelyEgress}(\frac{K}{2}, \frac{K}{2}+1, \ldots, K))$
2. if $E > THRESH_{HIGH} \times E_{cap}$ then
3. $\alpha_{F2R} = \min(\alpha_{F2R} \times AGGR, MAX_{\alpha_{F2R}})$
4. if $E < THRESH_{LOW} \times E_{cap}$ then
5. $\alpha_{F2R} = \max(\alpha_{F2R} / AGGR, MIN_{\alpha_{F2R}})$
6. return $\alpha_{F2R}$

Figure 6.2: Periodic adaptation of the ingress-vs-redirect control parameter.

$R$ stabilized), and it also takes time to react to this change and adjust the volume assigned to the server ($E + R$). Therefore, the adaptation routine in Figure 6.2 is called at certain periods. Every 20 minutes has shown long enough in our experiments; indeed the value of this and other parameters described below is CDN workload specific and is to be (re-)adjusted as per the following tunings. $THRESH_{HIGH}$ and $THRESH_{LOW}$ in the scheme determine when the tuning parameter $\alpha_{F2R}$ should start ascending or descending, for which we have found the relatively high values of respectively 0.95 and 0.9 necessary for maintaining fully utilized egress. The value of $AGGR$ (aggressiveness) defines the responsiveness of $\alpha_{F2R}$ to egress changes. Since too much fluctuations in the ingress-vs-redirect tradeoff is not desired, we opt for the smallest aggressiveness that is still fast enough in responding to load changes, resulting in a value of 1.5 from our different experiments. $MAX_{\alpha_{F2R}}$ bounds $\alpha_{F2R}$ so that it does not grow excessively if there is plentiful demand so that it can responsively shrink back and regulate redirects upon drop of egress—10 in our experiments. $MIN_{\alpha_{F2R}}$ defines the baseline operating point for off-peak times which is 1 for off-net servers (Line 5).

6.1.3 Experimental Results

We evaluate the control algorithm of Figure 6.2 on the traces of an actual server. Our dataset consists of server-based request logs and does not include the excess demand of the corresponding user network beyond the server’s capacity. Those requests are directly sent upstream and are never seen by the server. To overcome this problem and experiment with the ingress control algorithm as if we have some excess demand, we imagine an egress cap of 750 Mbps for the studied server in this case. Moreover, for this experiment we have implemented a simple traffic mapping component as well. This component receives feedback from the server on its egress and redirected traffic and controls the rate of the request traffic assigned to it such that its egress does not overflow (§ 6.1.1). The remainder of the request traffic is not given to the server and is discarded as if it was assigned directly to some upstream server location.
Figure 6.3: Cafe-based adaptive ingress control. Days 23–26 correspond to a long weekend. Data points represent aggregated 1-hour measurements. Gray shades show peak hours, measured as the time the server’s egress traffic reaches a \( THRESH_{HIGH} \) (95%) factor of the capacity (750 Mbps) until it falls below it; small fluctuations in between are ignored.

The results of this experiment are plotted in Figure 6.3. Traffic assigned to upstream either directly or indirectly through a redirection by the server is labeled as “Redirected or not seen” in this figure. The total traffic on the servers’ uplink is the sum of this value and cache ingress, which we examine shortly. The figure shows the last 10 days of a 30 day run of Cafe-based adaptive ingress control on the traces of an off-net server in the US with 8 TB of disk given.

In Figure 6.3, the ingress control algorithm starts to reduce cache ingress as soon as the egress traffic approaches capacity (the gray shades). The algorithm has throttled the ingress traffic during peak hours to as little as 0, specifically during the times when the demand is at its highest. The demand (not plotted separately) can be imagined as the sum of “Egress” and “Redirected or not seen” traffic. There exist a number of transient outlying points in the throttled ingress in Figure 6.3(b) which are due to drop of demand. The ingress gain comes with no significant effect on the server’s egress utilization: no significant drop of peak egress from Figure 6.3(a) to Figure 6.3(b). The server’s ingress ascends back up in Figure 6.3(b) once we get past the peak, and sometimes there even exists more ingress during off peak hours in Figure 6.3(b) than there is with the default (non-adaptive) operation in Figure 6.3(a). This is
because in the former case, there are more cache misses right after the peak than there is in the latter, as we had skipped cache-fills and the disk contents need to catch up. In other words, we effectively deferred part of cache-fills from peak to off-peak hours which is a highly desirable feature.

Note that the above reduction in peak-time ingress comes at the cost of more redirections to upstream. It is therefore necessary to evaluate the net saving in the total upstream traffic. We measure this traffic as the total of ingress and redirected traffic and the excess volume directly assigned to upstream. The results show that the upstream traffic at its 95%-ile is dropped by adaptive ingress control by about 13% in the above experiment. In other experiments with different egress cap values (500 to 1000 Mbps) and disk sizes (1 to 16 TB), similar savings have been observed, up to 17% off the 95%-ile upstream traffic. The xLRU algorithm, as expected from Section 4.7, is not as successful as Cafe as the underlying algorithm for limiting the ingress volume and can achieve a saving of only up to 5% in this traffic.

6.2 Layered Caches in a Server

Given the multi-Gbps amount of I/O for serving video workload, a servers needs to designate a part of its RAM as memory cache on top of the disks to mitigate their limited read bandwidths—evaluated quantitatively in Section 6.4. Moreover, flash storage with its declining cost can be an alternative or additional layer of caching to alleviate this problem while also avoiding the high cost of provisioning all-RAM caches. In addition, Phase Change Memory (PCM), a recent persistent-memory technology with significant performance gains over flash [52, 58], can be employed either in between RAM and flash or instead of them, as evaluated in Section 6.4. Thus, to analyze server provisioning it is necessary to analyze the performance of caches in a 2+ layered setting.

6.2.1 Mutual Effects of Layered Caches

Layering the caches alters the input pattern to them as shown in Figure 6.5 and discussed shortly, and it changes their behavior. To understand this change, let us first review the interactions of the layers in a server. The twofold interaction includes the effect of the lowest layer (disk) on the higher layers as well as the effect of each higher layer on the lower ones.

In a traditional stack of replacement caches, the request stream arrives at the layers from top to bottom, each leaving its miss stream for the next. On the other hand, in a video CDN cache some requests are redirected from the server if they are for video chunks too unpopular to live on the server—less popular than the least popular ones stored. The cache algorithm in the lowest layer (disk) decides whether a request should be admitted to or redirected from the server, since that is the largest one and
Figure 6.4: The interactions of cache layers inside a sample server with 3 layers.

holds the least popular content. Any interior cache layer can as well perform cache admission for its own layer: whether a missing chunk is to be stored on the layer or it is to be served directly from the lower layer. Though, the analysis in this section shows that plain LRU replacement suffices and is the preferred choice for the interior layers. Put together, as illustrated in Figure 6.4, in the life of a request it is first checked for admission by the disk layer cache algorithm, e.g., Cafe. If the requested video chunks are too unpopular to be cached on the disk, they are clearly not worth caching on the interior layers either and it does not make sense to unnecessarily run the request through them. The request is simply redirected. Our analysis shows shortly that this in fact benefits the caching efficiency of the interior layers. Notice that the disk cache admission algorithm sees all requests including those later served by a higher layer and not read from the disk, therefore it will never redirect a request if the data exists on any of the layers. If the request is admitted, it is sent to the highest layer for serving and possibly flows downwards including to the disk cache itself; this time only for possible cache-fill as it is already admitted.

In this system, first, cache admission leaves a fundamentally different input stream to higher layers, compared to a traditional cache stack where the whole request stream simply flows downwards. Second, adding a cache atop an existing one also leaves a changed input pattern to the lower one, as in a standard cache stack. These two effects are shown in Figure 6.5 for a sample two-layer cache fed with the traces of a real CDN server. The figure plots the popularity curve for the original request stream (labeled “Original input”), the stream of requests admitted by Cafe in the lower layer ("Post redirection"), and the miss stream of the higher layer ("Past higher layer"). The original request stream follows more or less the expected power-law pattern. The stream hitting the top layer, though, miss the tail which corresponds to unpopular content redirected by cache admission. The request stream arriving at the lower layer misses the head of the curve as well, which is popular content served at the higher layer. With either of these two effects on the input stream, a caching algorithm may or may no longer yield the same size-vs-efficiency behavior as its standalone version. Knowing this behavior is critical for deciding cache sizes. As our following analysis shows, the first change that is due to the lowest layer’s admission
control has a significant effect on the caching efficiency of the higher layers. The second change, adding a layer above an existing stack of caches (hiding requests from it), while fundamentally changing the efficiency of the existing layers, can be assumed to have only negligible effect on the caching efficiency of the overall system. On the other hand, behind the scenes the newly added layer takes a significant load off the lower layers.

### 6.2.2 Analysis of Cache Layering

We analyze the layering of caches by simulating a server under real workload in several different layering settings. We consider xLRU and Cafe caching schemes for video CDN servers as well as the standard LRU scheme. LRU, most widely used in today’s Web caches, can be employed in the interior layers although it does not support the requirements of our target cache servers. This is because an interior cache is only a protection layer and does not necessarily have to distinguish ingress and redirected traffic. It can also simply operate on the basis of single video chunks without having to make one serve-vs-redirect decision for a multi-chunk range request for chunks of different popularities: LRU can just fetch and cache each single cache-missed chunk from the lower layer. In case of a mixed cache hit and miss for a chunk range request at some interior layer, the original request may be broken into smaller chunk ranges (non-contiguous) when arriving at the lower layer. We let this naturally happen rather than enforcing continuity of requests getting past a layer (assuming this is doable without loss of efficiency); also note that in our traces with 2-MB chunks most requests span very few chunks. We also avoid the complexities of ensuring 100% inclusive cache contents from a layer to the next. As the experiments show next, these complications are not necessary.
Figure 6.6 shows the effect of the lowest layer’s admission control on the efficiency of a standard LRU cache at some higher layer. The LRU cache is operating on the request stream admitted by the server. Thus, the smaller the disk size (going from one curve to the next), the higher the fraction of videos considered unpopular whose requests are redirected. Consequently, the request stream to the LRU cache gets more and more exclusive of unpopular content. This yields higher caching efficiency at the LRU cache by up to 20%. That is, the efficiency curve of the LRU cache for a fixed user workload can no longer be estimated on its own and is a function of the underlying disk cache size and algorithm. This is a key consideration for the server provisioning problem.

To analyze the second effect, the addition of a higher layer cache on an existing one, we evaluate the operation of the three caches, LRU, xLRU, and Cafe, on top of each other (9 cases) with different sizes. An LRU as the bottom layer is to simulate cases such as a flash cache below RAM and is not an applicable candidate for the lowest layer (disk) as discussed. Our results show that the evaluation of beyond 2 layers is not needed.

Table 6.1 shows the effect of such a layering for a scenario where a 4-TB Cafe disk cache operates with and without a higher layer 1-TB LRU cache. The efficiency of the disk cache alone is dropped...
significantly from 83.8% to 69.8%, given the change in its incoming request traffic; see the capped head in Figure 6.5. However, the efficiency of the system as a whole—including the traffic getting past it—observes marginal change (0.5%). Behind the scenes, the read load on the disk cache is significantly reduced from \(1009 - 164 = 845\) Mbps to 367 Mbps.

Figure 6.7 expands this comparison to across different cache size combinations. It plots both the caching efficiency and the total ingress and redirected traffic getting past the server with and without the existence of a higher layer cache. The difference in either metric is marginal in most cases except a few extreme scenarios such as a large cache of 1 TB on top of another 1–2 TB cache. However, in all the realistic cases (top \(\leq 1/4\)th bottom) the difference that a higher layer cache makes is negligible (\(\leq 2\%\) in caching efficiency) and this marginal difference is in fact towards improving the caching efficiency of the system as a whole. Note that this is not due to the increase in the overall cache size when a higher layer is added, since most content on the top layer overlaps with the bottom. This (minor) effect, which only exists with an LRU on the top and not an xLRU or Cafe, can be explained as follows. The top layer LRU, once handling a request for chunk \(x\), hides the subsequent requests for \(x\) from the lower layer for as long as the top layer’s cache age—minutes to hours in these experiments. The videos that are watched a few times within a small window and then forgotten for a while will therefore not make their way into and pollute the lower layer cache. With an xLRU/Cafe on top, however, both the first and the second request will hit the lower layer cache: the first as a redirect (proxy) and the second as a fill.

In the experiments with an LRU cache atop xLRU, the same results as the LRU+Cafe case have been observed. An xLRU or Cafe cache as the top layer, as just explained, makes a yet smaller difference
Figure 6.8: The impact of adding a higher layer LRU cache on the ingress-vs-redirect control in the Cafe algorithm. The top and bottom cache sizes are 1 and 4 TB respectively.

(in most cases ≤ 0.1 in efficiency). Moreover, a standard LRU protecting another LRU makes the least of the differences, as one can intuitively expect from the least-recently used ordering behavior.

Finally, we note that the above measurements look at the overall caching efficiency. Though, our caching algorithms are capable of balancing the ingress and redirected traffic, e.g., across peak and off-peak hours. To examine the impact of additional cache layers on such balance, we evaluate in Figure 6.8 the Cafe algorithm across different operating points ($\alpha_{F2r}$) with and without an LRU layer on top; recall that this is the case with the most noticeable effect on efficiency. The figure shows that similarly to the overall caching efficiency, the higher layer has no considerable impact on the ingress-vs-redirect balance.

Summary. The findings on layered caches are summarized as follows.

- A server’s disk cache algorithm and size can significantly affect the efficiency of the higher layer caches. The efficiency of a caching scheme as a function of its size, which is the standard foundation of any cache provisioning study, cannot be evaluated on its own for the interior layers and is largely dependent on the disk cache beneath it. This is captured in our optimization framework.

- Adding a layer of smaller size (≤ half) on top of an existing cache, while obviously changing the input to the main cache and hiding many requests from it, either does not impact the overall caching behavior of the system or marginally improves it (≤ 2%). Also, this marginal difference only occurs from the lowest layer to the next and not from then on. Inside the system, the new layer takes a significant read load off the lower ones. This means, e.g., we can measure the expected cache-fills and redirects of a given disk cache size irrespective of the combinations of higher layers employed above it, though the disk’s I/O load depends to a large extent on those layers.

- While Cafe and xLRU are the suitable schemes for the server’s disk cache—and Cafe the most efficient one—as a higher layer above them the standard LRU is in fact the best choice rather than Cafe and
xLRU themselves. For yet higher layers above the LRU, the three algorithms make no difference, suggesting the simpler one (LRU) as the preferred candidate (among the three schemes we examined in this analysis for reasons explained).

### 6.3 Optimized Server Provisioning

This section presents the server provisioning optimization problem and our solution to it. The situation for a server deployment differs from location to location. In case of off-net servers, some ISPs may transfer all their upstream traffic through a *transit* provider, a.k.a., a parent ISP (§ 2.1). Internet transit is often billed by 95%ile bandwidth usage, a measure of peak traffic excluding outliers. Some larger ISPs, on the other hand, may have settlement-free *peering* with a CDN server location. In this case, upstream traffic towards the CDN may be assumed free up to a certain bandwidth limit. Excess traffic beyond this limit normally makes it way to the CDN server location through an alternative costly transit path. The same situations exist for on-net servers, where their upstream traffic either traverses the CDN’s backbone infrastructure or possibly Internet transit paths: either free, billed by Mbps, or a hybrid. We incorporate these considerations on network cost and its interaction with server configurations.

We consider the problem based on peak-time values. That is, we get peak-time traffic values to meet bandwidth constraints and we assume traffic pricing based on peak-time usage. The problem for the less common case of average-usage pricing can be structured and solved based on the same principles and is not studied separately for space limitation. We also assume that CPU is not a bottleneck tighter than the egress capacity when serving bulky traffic; this is based on our observations from a real-world video CDN. We also assume that the cluster of servers to deploy have the same configuration, for the concern of balanced load. Given content sharding across co-located servers (§ 2.2.1), the servers receive nearly the same request volume and yield the same caching efficiency and cache I/O.

#### 6.3.1 Notation for the Optimization Framework

Let $I$ denote the number of cache layers, usually 2 to 4 as explained, $c_i$ ($1 \leq i \leq I$) the cost per GB for the storage technology at layer $i$ where $i = 1$ is the top layer, $b_i$ and $z_i$ its read and write bandwidth in Mbps, and $x_i$ its size to be determined. This size may be provided, e.g., for the disk layer, as one disk drive or a series of them to boost the read/write bandwidth. We consider an array of same-size disks in the server’s tray, which will avoid imbalanced hit rates and loads across the disks. The number of drives is $k_i$, the size of each $x_i/k_i$, and the aggregated read (write) bandwidth $k_i b_i$ ($/k_i z_i$).

Let $e_{i, I}(x_i, x_I)$ be the expected caching efficiency of layer $i$ when given a size of $x_i$ and the admission
controller in the lowest layer is given a size of \( x_I \); see Figure 6.6. Let \( f_{i,I}(x_I, x_I) \) be the corresponding cache-fill-to-egress ratio (§ 2.2.4). We define \( f_{0,I}(\cdot) := 0 \) and \( f_I(x_I) := f_{I,I}(x_I, x_I) \) for convenience. As analyzed, caching efficiency and its breakdown to cache-fill and redirection can be assumed irrespective of higher layers. That is, \( f_{i,I}(x_i, x_I) \) is the fill-to-egress ratio of layers 1 through \( i \) as a whole as long as the layers are of reasonable relative sizes (\( x_j \leq x_{j+1}/2 \)). Out of some \( E \) Mbps volume of traffic egressed by the server, \( E(1 - f_{i,I}(x_I, x_I)) \) will get past the caches at layers 1 through \( i \) and hit layer \( i + 1 \); also recall that the interior layers had better use a plain replacement cache such as LRU (§ 6.2). Upstream traffic is free up to bandwidth limit \( B \) Mbps (possible peering/infrastructure), beyond which it incurs a cost \( C \) per Mbps. \( B \) may as well be 0. \( D_{\text{max}} \) denotes the total demand of the corresponding user networks during its peak times. The servers would usually get to full utilization during this time (§ 6.1) and we can assume \( y \leq D_{\text{max}} \).

The egress capacity of one server is \( E \) (e.g., 4 Gbps), and the total egress capacity to be determined for deployment is \( y \), i.e., the number of servers is \( y/E \). It is indeed not realistic to optimize \( y \) to any arbitrary value as we see shortly. Providing this capacity incurs a cost of \( w/\text{Mbps} \): the cost of a server divided by \( E \). The cost of a server can include the hardware cost of a server (excluding the cache layers) and the possible hosting premium the CDN may need to pay the hosting ISP or IXP, if non-zero. Realistic assignment of the different input parameters is described in Section 6.4.

### 6.3.2 Optimization Problem Formulation

The server provisioning problem is formulated as follows. Target variables to be found are bold faced: \( y, x_I \) and \( k_i \).

\[
\begin{align*}
\min & \quad w y + \frac{y}{E} \sum_{i=1}^I c_i x_i + \max \left\{ y \times f_I(x_I) + (D_{\text{max}} - y) - B, 0 \right\} \times C \times (P \times B) \\
\text{s.t.} & \quad E \times f_{i,I}(x_I, x_I) \leq k_i z_i (1 - \epsilon) \quad (6.2) \\
& \quad E \times (1 - f_{i,I}(x_I, x_I)) - E \times (1 - f_{i-1,I}(x_{i-1}, x_I)) \leq k_i b_i (1 - \epsilon) \quad (6.3) \\
& \quad \frac{1}{k_i z_i} E f_{i,I}(x_I, x_I) + \frac{1}{k_i b_i} E \left( f_{i-1,I}(x_{i-1}, x_I) - f_{i,I}(x_I, x_I) \right) \leq 1 \quad (6.4) \\
& \quad y \leq D_{\text{max}} \quad ; \quad x_I \leq x_{i+1}/2. \quad (6.5)
\end{align*}
\]

The objective function in Eq. (6.1) includes the cost of egress hardware and/or hosting premium \((w y)\), cache storage at each layer \((\sum c_i x_i)\), and the upstream traffic. This traffic includes the cache-fill and redirected traffic getting past the servers as well as any peak-time demand beyond the egress...
capacity. When egressing at capacity, the total cache-fill traffic of the servers is \( y f_i(x_1) \) and the total of redirected and excess traffic is \( D_{\text{max}} - y \). This traffic is free up to \( B \) Mbps and incurs a cost of \( C \) per Mbps after that, as calculated in Eq. (6.1). One may consider peering not entirely free since it needs provisioning and incurs cost, while one may take it as an orthogonal cost that will be there in any case. The formulation in Eq. (6.1) including/excluding the shaded term \((P \times B)\) assumes the former/latter case; in the latter case, \( B \) is a target variable rather than an input constant and \( P \) is the unit cost for provisioning peering bandwidth. We analyze both cases in our experiments. Moreover, we can introduce a marginal cost to Eq. (6.1) for employing smaller storage units, such as 1-TB disk drives rather than 4-TB, which we defer to Section 6.4.

The constraint in Eq. (6.2) ensures that we do not overflow the layer-\( i \) cache of a server with excessive writes under peak load, i.e., when egressing at \( E \) Mbps hence incurring a write load of \( E \times f_{i,I}(x_i, x_I) \). A \((1 - \epsilon)\) factor of the maximum write/read bandwidth \((k_i z_i / k_i b_i)\) is enforced in Eqs. (6.2) and (6.3) to ensure a minimum read and a minimum write bandwidth at all times.\(^5\) Eq. (6.3) ensures that the read load (excluding writes) stays within limit. This load equals the total cache reads at layers 1 through \( i \) minus those at layers 1 through \( i - 1 \). We can assume from Section 6.2 that the cache hit traffic of layers 1 through \( i \) equals what layer \( i \) alone would have achieved which is \( E \times (1 - f_{i,I}(x_i, x_I)) \). From this load, the cache hit volume of layers 1 through \( i - 1 \) is taken off. Recall that \( f_{0,I}(\cdot) := 0 \) and \( f_{I,I}(x_i, x_I) := f_I(x_I) \). We also note that the aggregate r/w bandwidth is more limited than simply \( b_i + z_i \), e.g., a disk being read at full bandwidth cannot write data and vice versa. Thus, in addition to the r/w bandwidth limits, we add the constraint in Eq. (6.4) where the sum of the write and read bandwidth utilizations cannot exceed 1. Eq. (6.5) includes our original assumptions for reasonable provisioning.

### 6.3.3 Optimization Solution

The formulated optimization problem has a rather complex form to be dealt with by standard linear/convex optimization libraries. However, we note the small number of optimization variables \((y, x_i, k_i)\) and the fact that an exact solution with arbitrary values for them is not going to be helpful anyway. The following make the problem tractable.

- Caches of RAM, disk or other storages can only exist at certain coarse-grained sizes, e.g., 1 TB disks.
- The egress capacity is limited in resolution by one server capacity, e.g., 4 Gbps. It may be further limited if there needs to be a fixed number of servers per rack (third-party made), though we do not make this assumption.

\(^5\)This is to avoid the potential overload in the two extreme ends of the r/w spectrum, e.g., \( \epsilon = 0.05 \) ensures that even if the model expects very small write loads for some scenarios we still bound read utilization to a maximum of 95% and always reserve a minimum 5% for writes.
• Not any arbitrary combination of possible cache size and egress capacity values is valid, e.g., a disk of 1 TB does not match an egress capacity of 10 Gbps (by server count).

• There exist no more than a few layers.

These restrictions on the solution space simplify the optimization problem: it can be solved by iterating over the solution space as we have easily done in our experiments. An example is as follows. There exist up to 4 cache layers and the lowest is disk. We are to deploy \( N \) servers at the target location, i.e., \( y = NE \). Suppose each server can egress up to \( E = 4 \) Gbps of traffic. Also, a server can hold up to 8 disks of \( \{1, 2, 4\} \) TB, giving the possible disk caches as \( \{1, 2, 4\} \times \{1, \ldots, 8\} \)—24 possible values for \( (x_4, k_4) \). Assuming about the same range of variability for the other 3 layers, only \( 24^4 \approx 330 \text{K} \) different combinations exist for each value of \( N \). In case of joint server and peering bandwidth provisioning discussed earlier, an additional dimension of the possible peering bandwidths \( (B) \) is added to the solution space with a similarly small number of values, e.g., by 100 Mbps–1 Gbps grains. The optimization has been solved in sub-second time in all our experiments.

**Optimization output.** In our solution process, we first fix the value of \( N \) (equivalently, \( y \)) and then optimize \( \{x_i, k_i\} \) values. In practice, CDN operators benefit more from such listed solutions that maps each \( N \) value to a vector \( \{x_i, k_i\} \) than from a fixed optimal value for all variables including for \( N \). This is specifically the case with off-net deployments. The listed solution, besides being a superset of the singular one, allows more informed decisions between a CDN and ISPs. In practice, the target number of off-net servers to deploy is not only a function of the corresponding hardware and bandwidth costs, but also largely a matter of each ISP’s willingness and ability to provide space for them. An ISP may either host any decided number of CDN servers on a settlement-free agreement (a mutual win), offer to host only a limited number, or charge the CDN a premium for hosting the servers—reflected in parameter \( w \). In any case, the quantitative figure provided by the optimizer on the resultant hardware and bandwidth cost for each given number of servers \( (N) \) is necessary for accurate and informed agreements between the CDN and each ISP.

### 6.4 Evaluation

In this section, we first review the involved parameters and the values used in our experiments. We then analyze the optimal configuration across a variety of scenarios and examine its relationship with

---

6The optimization problem can be solved fast with several times more combinations of storage sizes as well, as shown shortly. The choice of 32 TB as the highest disk size in our experiments is based on our experiments where >32 TB disk space would hardly warm up and reach a steady state during the duration of our traces. This yields a total disk size in \( \{1, 2, 4\} \times \{1, \ldots, 8\} = \{1, 2, \ldots, 8, 10, 12, \ldots, 16, 20, 24, \ldots, 32\} \): 16 values in \([1, 32]\).
different costs and constraints. As elaborated in Section 3.5, we are not aware of any previous work able to address the requirements of our studied provisioning task.

6.4.1 Evaluation Setup

Below, we revisit the input parameters for optimizing a server deployment (§ 6.3.1), how they should be set in the real world, and the values used in our experiments.

**Dollar cost $c_i$ per GB for the cache at layer $i$:** while the cost of equipping a server with 8 disks of 1 TB or 2 disks of 4 TB may not be exactly the same, it is common practice to measure storage cost per GB in the market, e.g., roughly 0.10 \$/GB for disk, 0.50 \$/GB for flash storage, and 10 \$/GB for RAM\textsuperscript{7}; PCM technology is estimated about 4× as costly as flash [52]. These values are summarized in Table 6.2. In our experiments, we made a minor modification to the objective function to prefer an obtained cache size ($x_i$) as larger storage units than smaller if yielding a similar objective function value and meeting all constraints, i.e., prefer smaller $k_i$. For example, assuming the 0.10 \$/GB cost for disk corresponds to its largest units (considered 4 TB in our experiments), we add 10\% to the disk cost if 2-TB disks are used and another 10\% for 1-TB disks. The corresponding modification to the objective function is trivial given the number of drives as $k_i$ and the size of each as ($x_i/k_i$), and it is omitted in Eq. (6.1) for clarity.

**Read and write bandwidths $b_i, z_i$ for layer $i$ cache:** although readily available as part of the hardware specifications, these numbers often need to be tuned. For example, up to 3 Gbps for random read and 1 Gbps for random write bandwidth has been measured across a number of Intel flash SSDs, far less than their specified sequential r/w performance\textsuperscript{8}. Hard disks usually have a read and write bandwidth of up to 1 Gbps for sequential r/w and multiple times less than that for random read and write. The numbers depend on the disk workload and the number of performed seeks. CDN operators often have a clear figure on these bandwidths based on their existing deployments. For the current evaluation, for each disk drive we consider a default of 400 Mbps read and 300 Mbps write bandwidth in some experiments and a range of bandwidths in others. If having multiple disks, their bandwidths add up. If multiple bandwidth/price configurations exist from different vendors, the optimization can be solved separately for each given that each optimization runs in sub-second time. RAM speed can be assumed infinite compared to any egress capacity. PCM technology is still in its early phases. A recent profiling of the performance of a PCM prototype reports up to \sim 21 Gbps sequential read and 2.4 Gbps sequential write bandwidth, while random 4-KB reads and writes are several times less efficient [52].

\textsuperscript{7}http://goo.gl/NW6dSH; http://goo.gl/Ud8bG6; http://goo.gl/d1ystW.
\textsuperscript{8}http://goo.gl/82xGyr
assume a PCM technology with 7 Gbps read and 800 Mbps write bandwidth as default and vary these numbers in some experiments. Note that 7 Gbps of read bandwidth exceeds the egress capacity but it cannot be assumed $\infty$ since PCM write is not as fast and the joint read and write bandwidth utilization is still a concern (Eq. (6.4)). Table 6.2 summarizes these default values.

Table 6.2: The default parameter values for storage technologies. Values vary in a range in some experiment. “M” represents Mbps and “G” Gbps in read/write bandwidth values.

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Write</th>
<th>$/GB</th>
<th>Sizes per slot</th>
<th>Max #</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>10.00</td>
<td>${16, 32, 64}$ GB</td>
<td>4 chips</td>
</tr>
<tr>
<td>PCM</td>
<td>7 G</td>
<td>800 M</td>
<td>2.00</td>
<td>${128, 256, 512}$ GB</td>
<td>2 chips</td>
</tr>
<tr>
<td>Flash</td>
<td>3 G</td>
<td>1 G</td>
<td>0.50</td>
<td>${0.5, 1, 2}$ TB</td>
<td>2 SSDs</td>
</tr>
<tr>
<td>HDD</td>
<td>400 M</td>
<td>300 M</td>
<td>0.10</td>
<td>${1, 2, 4}$ TB</td>
<td>8 drives</td>
</tr>
</tbody>
</table>

Peering/infrastructure bandwidth $B$ & transit cost $C$: unless a target variable to be found ($\S$ 6.3.2), bandwidth $B$ up to which the servers’ upstream traffic is free is known for each off-net or on-net deployment; $B$ may also be 0. Cost $C$ per Mbps for traffic beyond $B$ can represent the cost for only the CDN or both the CDN and the ISP, depending on the CDN operators’ decision and the terms of agreement with the ISP, e.g., connectivity for the hosted servers free of charge. In any case, the list of optimization outputs per each value of $N$ enables the CDN operators to sketch for each ISP its expected upstream traffic for the different server counts it can host ($N$), enabling an informed agreement. If the CDN desires no transit cost, it can enforce the optimizer to output egress and storage capacities matching this restriction by assigning $C = \infty$.

Egress capacity cost $w$ and peak-time demand $D_{MAX}$: the cost of egress capacity per Mbps ($w$) is simply the cost of a server divided by its egress capacity. The cost of a server can include two components. First, this cost may include the hardware cost of a full server (excluding the cache layers), or it may not: in some cases deploying a server at the new location is simply equivalent to saving almost the same hardware cost by freeing one on, e.g., the upstream on-net location. In some other cases, the CDN may have enough spare capacity on the on-net location, thus deploying the new servers does not save cost in terms of server hardware. Second, the server cost can include the possible premium to pay the hosting ISP or IXP, if non-zero. In addition, the peak traffic demand of the target user network(s) $D_{MAX}$ can be measured from the existing traffic, possibly projected to capture near-future growth.

Peak-time cache-fill-to-egress ratio: $f_{i,I}(x_i, x_I)$: this ratio determines the peak-time upstream traffic and the r/w load on each cache layer. Our experiments on different input workloads suggest that, given the importance of the accuracy of this function, it is best to be based on sample simulations on the actual traces of the target user networks rather than being estimated in theory.

For example, a rule of thumb for measuring the expected efficiency of a cache is through the workload’s
Figure 6.9: Cache efficiency of Cafe and LRU for different disk sizes as opposed to the estimate from the popularity curve of the workload. This estimate is labeled as “Static hitrate”.

The popularity curve, a sample of which is shown in Figure 6.5—assuming momentarily that we manage to derive the fill-to-egress ratio from cache efficiency and that the underlying cache admission and redirections can be theoretically taken into account when analyzing higher layer LRU caches. In this estimation, if the top 10% of video chunks make up 80% of the total requests, a cache of size equal to those 10% video chunks is estimated to achieve an efficiency of 80%, possibly times a data-fitting constant scaler. However, our experiments shown in Figure 6.9 indicate that neither the popularity-based estimate nor a scaled form of it can represent the actual efficiency in practice. In Figure 6.9(a), the efficiency of Cafe cache is up to 10% higher than the popularity-based estimate for smaller disk sizes, but falls below it for larger disks. The efficiency of LRU, though, is close. In Figure 6.9(b), on the other hand, the efficiency of LRU is far below the estimate and that of Cafe close to it. This effect is likely due to higher temporal locality in server A’s workload [14], i.e., requests for a video chunk arrive closer to each other rather than being evenly spread over time. The possible modeling of the expected efficiency (and fill-to-egress ratio) of a cache based on some characteristics of the workload, such as the popularity curve and temporal locality among others, is an interesting problem left for a future study.

For our purpose, given the importance of the reliability of the estimated values and the ease of the limited number of simulations required (see below), we provide the $f_{i,I}(x_i, x_I)$ function by simulating the servers on actual traces.

Figure 6.10 shows the values of the sought function $f_{i,I}(x_i, x_I)$ for both the Cafe-based disk cache and the LRU layers above. In Figure 6.10(a), rather than a gradual descent from 1 to 32 TB, a sharp drop of Cafe’s ingress to is observed by increasing disk size from 1 to 4 TB followed by a slower decrease from 4 TB to 32 TB disk. This is due to the adaptive ingress control feature (the Appendix). The
effectiveness of this feature in limiting ingress during peak hours is quickly boosted by increasing the
disk size from 1 to 2 and 4 TB. A 1 TB disk size is too limited to hold enough content to allow serving
near egress capacity without updating the cache contents (ingressing) as much. An extension to 2 and
4 TB allows the ingress control feature to significantly throttle cache-fills during peak hours (32% drop),
and the boosting effect dilutes for larger disks since the ingress-to-egress ratio is already pushed to <5%.

In Figure 6.10, bold points represent simulated points for the $f_{i,I}(x_i, x_I)$ function, based on which
any other value is interpolated in the remaining small ranges (between simulated points). Since log-scale
increments of $x_i$ values is sufficient for obtaining the $f_{i,I}(\cdot)$ curve, this function can be estimated for the
full range of valid inputs with a limited number of simulations. We use $2 \times$ increments of $x_i$ as shown
in Figure 6.10. The total number of required simulations were less than 100 which altogether take less
than a day to run on the traces on a commodity PC.

6.4.2 Experimental Results

We first examine how a server configured by the optimization process operates in practice. Let us
consider the optimization of a server’s cache layers with specifications from Table 6.2 in a scenario with
$D_{max} = 8$ Gbps peak-time demand and $B = 4$ Gbps peering bandwidth. We analyze a variety of other
configurations shortly. Server cost $w$ is irrelevant to optimization with a fixed number of servers (1 in
here). We assume $C = \$18/Mbps$ total transit cost based on a $0.50/Mbps$ estimated monthly cost for
the near future\(^9\) and a 3-year lifetime expectancy of a server deployment before replacement with newer
hardware. We also assume $\epsilon = 0.1$. The optimized output of this configuration is 8 TB of disk cache as
4 drives, 2 TB flash cache as 2 drives, and a 64 GB RAM chip. The choice of smaller disk drives made

\(\text{http://goo.gl/mSE1t}\)
Figure 6.11: The operation of the server configured by the optimizer’s output: 96 GB of memory cache, 2 flash drives of 2 GB, 4 disk drives of 2 TB.

by the optimizer, 2 TB rather than 4, is to sustain the read/write load. We have observed separately that an alternative 2 disk drives of 4 TB would violate the constraint in Eq. (6.4).

Figure 6.11 shows the operation of this server in a sample 10-day window. In Figure 6.11(a), the ingress traffic is throttled as egress approaches the capacity, which is due to the adaptive ingress control feature (the Appendix). Given the 8 Gbps peak demand, the optimizer has estimated 4250 Mbps peak-time upstream traffic for the optimal solution which is shown as a horizontal line in Figure 6.11(b). This traffic includes the server’s cache-fills and redirects and the excess demand beyond its egress capacity. Figure 6.11(c) shows the flash drives’ read and write loads as well as the considered bandwidth limits. The maximum read and write bandwidth of the 2 flash drives together are 6 and 2 Gbps from Table 6.2, thus enforced as 5.4 Gbps (falls outside the figure) and 1.8 Gbs given $\epsilon = 0.1$ in this experiment; see Eqs. (6.2) and (6.3). Figure 6.11(e) shows the disk drives’ read and write load and the given bounds.
Figure 6.12: Optimal disk space for each of the 8 servers and the resultant upstream traffic as transit cost ($C$) grows.

of $4 \times 400 \times (1 - 0.1) = 1440$ and 1080 Mbps, respectively. Figures 6.11(d) and 6.11(f) show the sum of read and write bandwidth utilization (Eq. (6.4)) whose bounded is 1 in the optimization. The figures show that the optimizer has arranged cache space and storage drives for disk and flash such that they are utilized close to their limits, particularly the flash drives.

Note that the transient spikes exceeding the bound of 1 specially in Figure 6.11(f) are not unexpected behavior. We computed the cache-fill-to-egress ratios—the data behind this optimization—based on average peak-hours measurements rather than the worst case. While the latter approach yields unnecessary over provisioning and under utilized disks most of the time, we allow momentary outliers to be handled by secondary mechanisms and we avoid such waste. The secondary mechanism is the arbitrary load monitoring routine existing in a typical production server, e.g., upon temporary disk/egress/CPU overload start redirecting some requests. In Figure 6.11(f), disk utilization exceeds 1 for less than 7% of the time and by an average magnitude of 0.1. That is, we allowed that an overall of 0.7% of requests observe an extra redirection.

In the next experiment, we analyze the tradeoff between network and storage cost. To isolate these two costs from factors such as free peering capacity and cache layer combinations, we suppose no peering bandwidth in the first experiment (i.e., full transit) and no cost for higher layer caches above disk. That is, disk r/w bandwidth is temporarily assumed high enough for the load so only the cost for disk space matters. Figure 6.12(a) plots the required amount of disk space as transit cost $C$ grows; see the curve labeled as zero peering bandwidth. The figure corresponds to the serving of 40 Gbps user demand by a large rack of 8 servers, each with up to 32 TB of disk. We optimize the server count as well shortly. Figure 6.12(b) shows the corresponding amounts of upstream traffic including the 8 Gbps excess demand. The noticeable pause in Figures 6.12(a) and 6.12(b) on disk sizes 8, 16 and 32 TB is
due to our discretization of the $f_{i,t}(x_{i},x_{t})$ function to these values and interpolating the in-between based on them (Figure 6.10(a)). Nevertheless, this loss of visual smoothness makes negligible difference ($< 0.5\%$) in the overall optimized cost. It is interesting to note that if we consider the original r/w/ bandwidth limits of disks rather than $\infty$, higher cache layers become necessary whose mixed cost affects the cache-vs-network cost tradeoff. For example, for a transit cost of $C = 200$, the prescribed disk sizes reduce from 32 TB (Figure 6.12(a)) to 20 TB protected by 4 TB of flash and 32 GB of memory cache.

While the above experiment with all-transit upstream traffic helps isolating disk space versus network cost, in a more common scenario there would exist some peering bandwidth to make up for the 8 Gbps excess demand as well as the servers' cache-fill and redirected traffic. We therefore consider $8+1=9$ Gbps of peering bandwidth in the next experiment and evaluate how it shapes the optimal disk space. The dotted curves in Figure 6.12 represent this experiment. The optimizer shows that increasing the disk size per server to beyond 12 TB is not necessary for a transit cost of up to $150$, i.e., $4$/Mbps per month. This is because the resultant upstream traffic is almost within the 9 Gbps peering bandwidth and exceeds it by only 20 Mbps. If the transit cost continues to grow beyond $150$, the optimizer adds another 2 TB to the disk space to further cut that 20 Mbps of traffic. These experiments show how the right disk size to provision is a function of the existing peering bandwidth. If this bandwidth was below 8 Gbps in this experiment, we had to keep adding disks as transit cost grew; if it was beyond 10 Gbps, there would barely be any need for larger disk space than 2 to 4 TB. The joint optimization of cache servers and peering bandwidth is analyzed at the end of this section.

We analyze when and to what size each storage technology is cost effective as a cache layer based on its read/write bandwidth and price. Table 6.3 shows sample results based on one server, no peering bandwidth and 4 Gbps demand. Peering is analyzed next. As for the demand, note that excess demand beyond the server’s capacity of 4 Gbps does not affect the optimization results and only offsets the upstream traffic by a constant. As for the server count, in our model co-located servers have the same configuration ($\S$ 6.3.1) and the content ID space is sharded across them ($\S$ 2.2.1). Therefore, cache sizing results of the one-server case in Table 6.3 also apply to higher server counts when the demand and peering (0) are also proportionally scaled: only the last two columns will scale proportionally as we both observe from Eqs. (6.1) to (6.5) and have verified in our experiments.

Table 6.3 shows that if the r/w bandwidth of disk drives reduce by half, a higher number of smaller disks are prescribed to make up for the bandwidth deficit. The resultant optimal cost grows marginally which is for the 10% penalty we considered for small size disk drives. If the disk bandwidth doubles, though, the optimal configuration changes more radically to 6 (rather than 4) disk drives of 2 TB, whose aggregate throughput eliminates the need for higher layer caches in this scenario. This significantly
Table 6.3: Changes in bandwidth and price based on which a storage technology becomes cost ineffective for employment as a cache layer. The bold-faced row represents the base configuration. The third row reads as follows: if the read and write bandwidth per disk drive were 200 and 150 Mbps instead of the defaults of 400 and 300 Mbps, the optimal configuration would be 8 TB of disk cache as 8 drives, 2 TB of flash cache above it as 2 drives, no PCM cache, 64 GB of memory, and this would yield a total cost of $9,410 and an upstream traffic of 152 Mbps. Server hardware cost is assumed $4000 ($\sim \$1/\text{Mbps}) and transit cost $18/\text{Mbps}.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Disk</th>
<th>Flash</th>
<th>PCM</th>
<th>RAM</th>
<th>Opt. cost</th>
<th>Upstream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default (Table 6.2)</td>
<td>8 TB /4</td>
<td>2 TB /2</td>
<td>0 /0</td>
<td>64 GB /1</td>
<td>$9,410</td>
<td>152 Mbps</td>
</tr>
<tr>
<td>Disk r/w → 200/150 Mbps</td>
<td>8 TB /8</td>
<td>2 TB /2</td>
<td>0 /0</td>
<td>64 GB /1</td>
<td>$9,500</td>
<td>152 Mbps</td>
</tr>
<tr>
<td>Disk r/w → 800/600 Mbps</td>
<td>12 TB /6</td>
<td>0 /0</td>
<td>0 /0</td>
<td>0 /0</td>
<td>$7,697</td>
<td>130 Mbps</td>
</tr>
<tr>
<td>Disk $/GB: 0.10→0.02</td>
<td>16 TB /4</td>
<td>4 TB /2</td>
<td>0 /0</td>
<td>16 GB /1</td>
<td>$8,633</td>
<td>115 Mbps</td>
</tr>
<tr>
<td>Flash r/w 3/1→4/1.5 Gbps</td>
<td>12 TB /6</td>
<td>1 TB /2</td>
<td>0 /0</td>
<td>0 /0</td>
<td>$8,317</td>
<td>130 Mbps</td>
</tr>
<tr>
<td>Flash r/w 3/1→2/0.7 Gbps</td>
<td>8 TB /8</td>
<td>0 /0</td>
<td>0 /0</td>
<td>192/3</td>
<td>$9,654</td>
<td>152 Mbps</td>
</tr>
<tr>
<td>Flash $/GB: 0.50→0.3</td>
<td>12 TB /3</td>
<td>4 TB /2</td>
<td>0 /0</td>
<td>0 /0</td>
<td>$8,803</td>
<td>130 Mbps</td>
</tr>
<tr>
<td>Flash $/GB: 0.50→0.7</td>
<td>8 TB /8</td>
<td>0 /0</td>
<td>0 /0</td>
<td>192 GB /3</td>
<td>$9,654</td>
<td>152 Mbps</td>
</tr>
<tr>
<td>PCM write → 1.3 Gbps</td>
<td>8 TB /8</td>
<td>0 /0</td>
<td>512/2</td>
<td>32/1</td>
<td>$9,212</td>
<td>152 Mbps</td>
</tr>
<tr>
<td>PCM 1.3 write &amp; $/GB → 0.80</td>
<td>12 TB /6</td>
<td>0 /0</td>
<td>1 TB/2</td>
<td>0 /0</td>
<td>$8,516</td>
<td>130 Mbps</td>
</tr>
<tr>
<td>RAM $/GB: 10.00→8.00</td>
<td>8 TB /8</td>
<td>0 /0</td>
<td>0 /0</td>
<td>192 GB /3</td>
<td>$9,270</td>
<td>152 Mbps</td>
</tr>
<tr>
<td>RAM $/GB: 10.00→14.00</td>
<td>12 TB /3</td>
<td>4 TB /2</td>
<td>0 /0</td>
<td>0 /0</td>
<td>$9,622</td>
<td>130 Mbps</td>
</tr>
<tr>
<td>Transit $/Mbps: 18→7</td>
<td>5 TB /5</td>
<td>2 TB /2</td>
<td>0 /0</td>
<td>16 GB /1</td>
<td>$7,732</td>
<td>256 Mbps</td>
</tr>
<tr>
<td>Transit $/Mbps: 18→28</td>
<td>12 TB /3</td>
<td>4 TB /2</td>
<td>0 /0</td>
<td>0 /0</td>
<td>$10,925</td>
<td>130 Mbps</td>
</tr>
</tbody>
</table>

reduces the total cost by $1,700. Though, this amount of disk space would not be cost effective with the default disk bandwidth because of the higher read load per disk, thus requiring larger caches in the higher layers. Our experiments show that more expensive disk prices do not change the optimization result unless (trivially) if exceeding flash price—not shown in the table. That is, a multifold increase in disk price from 0.1 to 0.5 $/\text{GB}$ is still not worth reducing the disk space and incurring the more expensive cost of transit upstream traffic. Reducing the disk price similarly does not affect the optimization output unless if by several times, as shown in the table.

There will be no need for a memory cache if flash drives can read and write $\sim 30$–$50\%$ faster than we considered. This would allow distributing the load of memory cache in part on more disk drives (6) and in part on the now-faster flash drives. Slower flash drives, on the other hand, would require higher volumes of memory cache protecting them which was not as cost effective as just distributing the load of the flash layer across a higher disk count (8) and more memory space (192 GB), as shown in the table. Flash price per GB governs a tradeoff between how much flash or RAM to employ, if any. Only a 40\% increase in this price would render flash ineffective altogether, and a 40\% decrease makes an all-flash cache atop disk with no RAM the preferred choice. The results also show that PCM SSDs could be a replacement for flash drives if the problem with their limited write bandwidth did not exist—if they could write at least 1.3 Gbps in the current scenario. With 1.5 Gbps write speed, they could replace the memory cache as well (not shown in the table). If in addition to higher write throughput (1.3 Gbps),
PCM chips also reduce in price from $2/GB to $0.80/GB or less, there will be no need for a memory cache. The table also shows that less expensive RAM by only 20% or more would eliminate the need for any lower layer cache until disk, while more expensive RAM by 40% or more enforces larger flash drives to make the expensive memory cache unnecessary. Transit traffic cost closely shapes the optimal disk space, as analyzed earlier, and the necessary protection layers above it as shown in the table. A transit cost of $7 or less shrinks the prescribed disk space from 8 to only 5 TB, and a cost of $28 increases it to 12 TB in order to reduce transit traffic from 152 to 130 Mbps.

Finally, we analyze the provisioning of peering bandwidth; that is, where $B$ is a target variable to be found in Eq. (6.1). The cost of provisioning ($P$) can be either that of installing routers and cables or renting peering ports at IXPs. In the case for on-net servers, this is the cost for providing bandwidth infrastructure in the CDN’s backbone. In this experiment, we assume the same 40 Gbps peak demand and would like to find the right peering bandwidth the CDN should have with an ISP based on the joint cost of peering provisioning and server provisioning. In the first scenario, we assume the unit cost for provisioning peering bandwidth is $P = $2/Mbps: about an order of magnitude less that purchasing transit ($C = $36/Mbps). We also assume a $4000 per-server price (i.e., $w \approx $1/Mbps) including server hardware (excluding cache layers) and/or server hosting premium ($\S$ 6.3.1).

Figure 6.13(a) plots the optimal costs for this scenario with each possible server count as a separate curve. Focusing on each curve individually, we see that the optimal peering bandwidth for the respective server count is where the curve hits its minimum: a peering bandwidth just enough to accommodate the servers’ upstream traffic and the excess demand. Any smaller peering bandwidth to the left of the optimal point results in rapidly growing transit cost, and any larger peering bandwidth is over-provisioning and a (linearly increasing) waste of budget with no caching efficiency gain. Moreover, comparing the optimum
across the curves shows how smaller server counts result in higher peering bandwidths necessary. It also shows the important tradeoff between provisioning more servers or more peering bandwidth to an ISP. In the current scenario (Figure 6.13(a)), providing 10 servers and 3 Gbps of peering is the optimal choice—assuming the ISP is willing to host that many.

This tradeoff is inherently shaped by the relative price of peering provisioning and server cost—\( P \) and \( w \). For example, Figure 6.13(b) redraws Figure 6.13(a) when we manage to provide peering at half price (\( P = $1/\text{Mbps} \)). In this case, providing more peering bandwidth than physical servers and cache layers is the cost effective approach. Further experimentation with lower and higher peering costs (\( P \)) show that with a fixed server cost of \( w \approx $1/\text{Mbps} \) (as per $4000 per 4 Gbps) and the storage costs listed in Table 6.2, for \( P \geq $1.6/\text{Mbps} \) it is always better to deploy the maximum number of servers (10) for the 40 Gbps demand; for \( P \leq $1.4/\text{Mbps} \) it is best to deploy the least number of servers (1) and instead install more peering for the excess demand; \( P \approx $1.5/\text{Mbps} \) balances the cost of peering and server provisioning across different server counts—the CDN operators could go either way.

6.5 Summary

This Chapter addressed the problem of configuring the proper CDN server cluster for each server location. First, an adaptive ingress control scheme for minimizing a server cluster’s peak upstream traffic was presented, which can reduce the servers’ ingress traffic (and disk write) to close to 0 and their total upstream traffic by up to 17% specifically during periods of peak load and excess demand; this reduces the network bandwidth and disk size to provision. Moreover, the interaction of cache layers inside each server, such as SSD and RAM above the disks, was analyzed using actual server traces. This showed that the behavior of a caching algorithm can change significantly depending on its lower layers, but it changes only marginally by the addition of a higher layer which in turn will take a significant read load off the lower layer. An optimization framework was designed to find the right server configuration and server count (and peering bandwidth) to deploy at a given location. Extensive experimentation verified the validity of the prescribed configuration and drew several insights on the optimal configuration along a variety of parameters involved, including transit bandwidth cost, peering capacity, and storage (disk, flash, PCM and RAM) read/write bandwidths and prices.
Chapter 7

Minimum-Delay Message Distribution

In this chapter, we develop a suite of algorithms for the generic problem of distributing delay-sensitive messages to large groups of receivers, a problem underlying several distributed applications: an event generated at a node needs to be signaled to a large group of monitoring nodes with minimum latency. Some of the examples include our targeted content distribution networks in the earlier chapters as well as their peer-assisted descendants [43, 61], message distribution to all ends (counting even up to millions) in online applications based on shared data [8], massive multiplayer online games [54, 46], financial trading through large groups of globally connected computing hardware [78, 1], and distributed interactive simulation software [34, 41]. Also note that the group of receivers monitoring each source node in these systems may not be constant over time, such as a dynamic agent (player) in a virtual environment (online game) moving across the area of interest of other entities [11, 24], or a video being watched by a variable set of users.

To distribute messages, forming fixed overlay multicast groups (which nodes should join and leave) in such dynamic systems and maintaining the corresponding state information in the intermediate overlay nodes, as in several classic multicast techniques [85, 95, 42, 56], is not an efficient solution. A naive alternative approach is to send each message directly from the source to each receiver, which is not scalable as it requires each node to have (and constantly monitor the state of) a connection to every other node in the network. Moreover, this naive approach can incur long delays, because a node has a finite-bandwidth connection to the network, over which several copies of the same data should be sent. To avoid these problems, nodes can be connected through a dynamic mesh overlay. Then, source-based
minimum-delay multicast trees can be built on demand by the source and embedded in the messages (e.g., using Bloom Filters [48, 84]). This is the distribution model of our interest in this chapter.

While the primary goal of typical shortest-path multicast trees has been to minimize the total link-by-link delay experienced by packets, we have observed that in application-layer multicast a significant portion of the total delay is the delay incurred at overlay nodes. This is because each intermediate node has to send out several copies of the same packet through a single, finite-capacity network connection (Figure 7.1). This node-incurred delay, which is in proportion to the degree of the node in the routing tree, is in addition to the delay occurring in overlay links and can even dominate it as we show shortly. In particular, this issue becomes more critical in large-scale overlay networks with strong connectivity. In these networks, as intuitively expected, most shortest paths consist of a few hops only [101, 98], leading to large node degrees in a multicast tree.

To get a numeric intuition on this problem, suppose we would like to deliver one packet of 1.5 KB (12 Kbits) to 1000 nodes in an event notification overlay. Assume that overlay nodes have a 10 Mbps upload bandwidth and are on average forwarding messages of up to 5 concurrent sessions; hence, it takes about 6 ms for a node to send out one copy of the packet. Also, assume that the delay between every pair of nodes is 82 ms (according to our dataset [96]), and the average shortest path length is 3 hops in the overlay (i.e., a delay of 246 ms). Thus, the average degree of nodes in the multicast tree is about $1000^{1/3} \approx 10$, and the average delay incurred by each node to forward the packet is $\text{average}_{i=1,...,10}(i \times 6 \text{ ms})=33 \text{ ms}$, i.e., a total node-incurred delay of 99 ms in a typical 3-hop path which is more than 40% of the total link-by-link delay of such path (246 ms).

Yet, the problem with delays incurred by node degrees in application-layer multicast is either ignored [16, 33, 60, 15, 73] or only partially addressed by previous works, such as bounding node degrees in a tree to predefined thresholds [19, 80, 22, 20, 99]. The problem with large node degrees, however, is of the same type as the shortest-path routing problem—minimizing the incurred delay. It thus needs to be considered together with link delays in the routing decisions, rather than as a separate problem and at the coarse grain of being or not being within a threshold.
We study the overlay multicast routing problem for minimizing the actual end-to-end delay. We first formulate the two problems of minimizing the average and the maximum delay in multicast trees, and we prove their NP-hardness as well as their inapproximability to within any reasonable ratio in Section 7.1. That is, we show that no polynomial-time approximation algorithm can guarantee any better delay than several times beyond the optimal value. We then develop a set of efficient algorithms for building multicast trees with minimum average (or minimum maximum) delay in Section 7.2. These algorithms support a wide range of applications: different overlay scales, real-time requirements and session characteristics. To demonstrate the effectiveness of these algorithms, we conduct in Section 7.3 an extensive evaluation study on different real-world datasets, using three different overlay network models. We show that the actual delay observed by multicast receivers can be reduced by up to 60% and the calculation time for multicast trees by up to orders of magnitude (i.e., supporting several times larger scales), if our algorithms are employed.

### 7.1 System Model and Problem Statement

This section presents the formal statement of our multicast problems and their hardness, followed by a description of the routing model underlying our algorithms.

#### 7.1.1 Notation for Multicast Problems

A summary of the notations used in this chapter is given in Table 7.1. Consider an overlay network interconnecting a large population of distributed hosts. The overlay is modeled by a graph $G = (V, E)$.
where $V$ and $E$ represent hosts and overlay links, respectively. Let $N = |V|$, and $w(u, v)$ be the length of edge $(u, v)$, which is the network delay between nodes $u$ and $v$ in our application. We assume each overlay node is connected to the underlying network, typically the Internet, via one link (nodes are assumed not multi-homed; also a node with multiple NICs connected to the same network can be modeled as having one NIC with the aggregated bandwidth). The connection bandwidth of each node is a finite number, according to which we define $\Delta_u(z)$ as the time it takes for node $u$ to send $z$ units of data to the network. Since the time for node $u$ to process a message is much smaller than the time it takes to send out (possibly multiple copies of) the message to the network, we ignore the processing time at $u$ and let $\Delta_u(\cdot)$ be a function of the connection bandwidth of $u$ only. For example, for a host with a 10 Mbps Internet connection and a message of $z = 1$ bit, $\Delta_u(z) = 10^{-7}$ seconds.

We capture the node degree-incurred delays (also referred to as nodal delays) in a multicast tree as follows. Let $T$ denote an arbitrary multicast tree rooted at a given source $s$ and reaching the receiver set $R \subseteq V$, and $d_T(u)$ be the number of children of $u$ in $T$. Once $s$ started distributing a message of size $z$ at time 0, $t_T(v)$ is the time at which node $v$ receives the message over $T$. Assuming node $u$ is the parent of $v$ in $T$, we have:

\[
\begin{align*}
t_T(s) &= 0 \\
t_T(v) &= t_T(u) + w(u, v) + \Delta_u(z) \times q_{u,T}(v),
\end{align*}
\]

(7.1)

where $q_{u,T}(v)$ ($1 \leq q_{u,T}(v) \leq d_T(u)$) shows the turn of node $v$ among the $d_T(u)$ children of $u$ in $T$ that are waiting to receive a copy of the message.

Because in some cases we may not be able to explicitly dictate an order among the children of $u$ in $T$, we also take the expected delay observed at each child of $u$ into account. We define $\hat{t}_T(v)$ as in the following equation, by replacing $q_{u,T}(v)$ in Eq. (7.1) with $E[q_{u,T}(v)]$, the average of possible turns between 1 and $d_T(u)$ for a child.

\[
\begin{align*}
E[q_{u,T}(v)] &= \frac{1}{d_T(u)} \sum_{i=1}^{d_T(u)} i = (d_T(u) + 1)/2 \\
\hat{t}_T(s) &= 0 \\
\hat{t}_T(v) &= \hat{t}_T(u) + w(u, v) + \Delta_u(z) \frac{d_T(u) + 1}{2}.
\end{align*}
\]

(7.2)
7.1.2 Formal Definition of Problems

We consider two versions of the minimum-delay multicast problem: (i) Constructing a multicast tree in which the average delay (equivalently, the total delay) is minimized, and (ii) a tree in which the delay to the farthest node is minimized. We propose algorithms for both versions of the problem.

Our algorithms for case (i) are not based on encoding the tree structure in full in the data; they do not need to specify a forwarding order for each intermediate tree node, and can work with \( \hat{t}_T(u) \) values defined in Eq. (7.2). This is preferred, as it enables the use of mechanisms such as Bloom Filters for encoding the tree [48, 84]. The algorithms for case (ii), on the other hand, have to specify a forwarding order for the children of each node (and therefore work with \( t_T(v) \) values defined in Eq. (7.1)), since the maximum delay in a tree depends to a large extent on the ordering of each node’s children.

Our minimum-delay multicast problems can be formalized as follows.

Problem 3. Given a source node \( s \) and a set of receivers \( R \subseteq V \), construct a multicast tree \( T \) such that

\[
\sum_{u \in R} \hat{t}_T(u) \text{ is minimized.}
\]

Problem 4. With the same inputs as in Problem 3, construct a multicast tree \( T \) such that \( \max_{u \in R} t_T(u) \) is minimized.

The total expected delay in a multicast tree, \( \sum_{u \in R} \hat{t}_T(u) \), can be rewritten as the following for simplicity in the theorem below and our algorithms. Let us define an auxiliary cost function, \( f_T(u) \), for each node \( u \) which is the total expected delay (in terms of \( \hat{t}_T(\cdot) \) values) in the subtree rooted at \( u \) in \( T \), and let \( g_T(u) \) be the number of receivers in this subtree. We have:

\[
c_T(u, v) := w(u, v) + \Delta_u(z) \times q_{a,T}(v) \\
f_T(u) = \sum_{v \in \text{children}_T(u)} \left( f_T(v) + g_T(v) \cdot c_T(u, v) \right) \\
\sum_{u \in R} \hat{t}_T(u) = \sum_{v \in \text{children}_T(s)} \left( f_T(v) + g_T(v) \cdot w(s, v) \right) + |R| \times z \times \Delta_s(z) \times \frac{d_T(s) + 1}{2}. \tag{7.3}
\]

Theorem 2 shows the NP-hardness as well as an inapproximability factor for Problems 3 and 4.

Theorem 2. Problems 3 and 4 are NP-hard. No polynomial time approximation algorithm for either of them can guarantee an approximation factor of \((1 - \epsilon) \ln n\) for any \( \epsilon > 0 \) (under the conventional assumptions for \( P \) and \( NP \)).

Proof. The proof for both problems is based on reduction from the set cover problem. An instance \((U, S)\) of the set cover problem consists of a universe \( U \) of elements \( u_j \in U \) \((1 \leq j \leq |U|)\) and a collection \( S \) of
Chapter 7. Minimum-Delay Message Distribution

subsets $S_i \subseteq U$ ($1 \leq i \leq |S|$) whose union is $U$. The goal is to find a minimum number of subsets whose union covers the entire $U$. We can transform an instance of the set cover problem to an instance of the overlay multicast problem as follows. Each subset $S_i \in S$ is represented by a node $x_i$ ($1 \leq i \leq |S|$) in the overlay, and each element $u \in U$ is represented by a node $y_j$ ($1 \leq j \leq |U|$). Figure 7.2 illustrates an example. An additional node $s$ is added to the tree as the source node. There is an edge from $s$ to all $x_i$, and an edge from $x_i$ to $y_j$ iff subset $S_i$ includes the element $u_j$; see the example in Fig. 7.2. We also assign:

$$R = \{y_1, \ldots, y_{|U|}\}, \Delta_s(z) = 1, \Delta_{x_i}(z) = \Delta_{y_j}(z) = 0,$$

$$w(s, x_i) = w(x_i, y_j) = 0 \ (1 \leq i \leq |S|, 1 \leq j \leq |U|).$$

One can easily verify that given a multicast tree $T$ on this network that either minimizes $\sum_{u \in R} \hat{t}_T(u)$ or $\max_{u \in R} t_T(u)$, the set of nodes $x_i$ that are included in $T$ gives the solution to the set cover problem (consisting of $d_T(s)$ subsets). Hence, Problems 3 and 4 are NP-hard.

To establish the inapproximability bound, we first note that according to Eq. (7.3) the total and the maximum delay of a multicast tree $T$ on this network for distributing a unit-size message can be calculated as follows:

$$\sum_{u \in R} \hat{t}_T(u) = f_T(s) = \frac{d_T(s) + 1}{2} \times |U| \quad (7.4)$$

$$\max_{u \in R} t_T(u) = \max_{v \in \text{children}_T(s)} q_{s, T}(v) = d_T(s). \quad (7.5)$$

Suppose we are given an $\alpha$-approximation algorithm ($\alpha > 1$) for Problem 3, which can build a multicast tree $T$ in which the total distances to all nodes is at most $\alpha \times OPT$, where $OPT$ is the total
Chapter 7. Minimum-Delay Message Distribution

distance in the optimal tree, $T_{OPT}$. Hence, according to Eq. (7.4), we have:

$$\sum_{u \in R} t_T(u) \leq \alpha \times OPT \Rightarrow d_T(s) + 1 \leq \alpha(d_{T_{OPT}}(s) + 1)$$

$$\Rightarrow d_T(s) \leq \alpha d_{T_{OPT}}(s).$$

Similarly, given an $\alpha$-approximation algorithm for Problem 4 and assuming $OPT$ is the maximum distance in the optimal tree for this problem, $T_{OPT}$, according to Eq. (7.5) we have:

$$\max_{u \in R} t_T(u) \leq \alpha \times OPT \Rightarrow d_T(s) \leq \alpha d_{T_{OPT}}(s).$$

Thus, in both cases we could find an $\alpha$-approximation algorithm for the set cover problem. On the other hand, given the well known $(1 - \epsilon) \ln n$ inapproximability of the set cover problem in polynomial time [36], it is clear that Problems 3 and 4 cannot be approximated in polynomial time to within a factor of $(1 - \epsilon) \ln n$ for any $\epsilon > 0$, since otherwise a contradiction results.

7.1.3 Routing Model

To enable a fully distributed routing scheme, overlay nodes need to generate and exchange periodic routing information. Link-state routing protocols, which are commonly used for overlay routing [56, 15, 16, 48, 101, 73], enable each node to know the full network topology by getting each node to broadcast the state of its neighboring links to the entire network. This scheme, however, has limited scalability due to its high overhead: $O(D^2 N)$ per node\(^1\) where $D$ is the average degree in the overlay; each node generates a message of size $O(D)$ which traverses $O(|E|) \simeq O(DN)$ links. On the other hand, distance-vector routing used in [33, 60] incurs a lower overhead since each node announces to its neighbors only its shortest distance to other nodes. The corresponding overhead is on average $O(DN)$ per node—up to orders of magnitude more scalable than link-state routing in well connected overlays. However, nodes will have no information about the paths other than the next hop and the total length. It is thus not possible for a source node to customize a multicast tree. Distance-vector routing can also suffer from routing loops and long convergence times.

Therefore, we adopt a modified version of distance-vector routing, where in addition to the shortest distance to each destination, a node also announces the path itself (in our case the nodes also announce their $\Delta_u(\cdot)$ values along with this information). This is similar to the technique employed in the BGP protocol and is usually referred to as path-vector routing. This approach allows each node $u$ to construct

\(^1\)These rough estimates are used for a first assessment only.
a graph \( G^u_{PV} = (V^u_{PV}, E^u_{PV}) \) where \( V^u_{PV} = V \), and \( E^u_{PV} \) consists of only a representative subset of \( E \) for \( u \): The edges on the shortest path, as well as up to \( d_G(u) - 1 \) alternative short paths, from \( u \) to all destinations in the graph. The overhead for exchanging path information is on average \( O(DLN) \) per node, where \( L \) is the average shortest-path length in the overlay. Note that \( L \) (which is upper-bounded by the overlay diameter) is a small number in a well connected overlay \([91, 10]\); 5–10 hops in all of our experiments with different overlay models and \( N = 100, \ldots, 4000 \) nodes. Consequently, we can inherit much of the scalability of distance-vector routing since the \( O(DLN) \) overhead is much closer to that of distance-vector routing \( (O(DN)) \) as opposed to link-state routing \( (O(D^2N)) \).

Moreover, path-vector routing does not suffer from routing loops and long convergence time as in the distance-vector approach, since the path info is given. Our multicast algorithms can indeed work on top of link-state routing protocols as well, in scenarios where the higher overhead is not a concern.

### 7.2 Minimum-Delay Multicast Algorithms

In this section, we first provide an overview of our minimum-delay multicast tree algorithms, and then present the details as well as the analysis of the algorithms.

#### 7.2.1 Overview of Algorithms

Our algorithms include two operation modes: MinSum for minimizing the expected total delay (Problem 3) and MinMax for minimizing the maximum delay in the tree (Problem 4). We refer to these algorithms as MSDOM and MMDOM (MinSum/MinMax Delay Overlay Multicast). These algorithms outperform the previous related approaches in both tree efficiency and running time, as analyzed in the next section.

Nevertheless, to further extend the application of our work to larger overlays, we design an additional algorithm for each operation mode (MinSum/MinMax) that is optimized for speed, with orders of magnitude faster running times. These algorithms are particularly suitable for large overlays where our former algorithms (and the related previous approaches) cannot operate fast enough. We refer to the former (delay-efficient) algorithms as MSDOM-\( e \) and MMDOM-\( e \), and to the latter (fast) version as MSDOM-\( f \) and MMDOM-\( f \) algorithms.

In addition, we design an algorithm for building a new multicast tree by only updating a previous one in nearly zero time. This algorithm leverages the fact that the receivers in a source node’s multicast group usually change gradually, rather than being shuffled from each message to the next. Thus, instead of rebuilding a whole tree from scratch every time, our new algorithms which we call iMSDOM and
Chapter 7. Minimum-Delay Message Distribution

iMMDOM, *incrementally* build a multicast tree by adapting a previously created (outdated) tree according to the recent changes in the receiver group. We also demonstrate in Section 7.3 how these algorithms can be best employed on top of our other algorithms to always yield the best running time and tree efficiency in practical scenarios. A summary of our different algorithms and their capabilities is given in Table 7.2.

Each overlay node that is to distribute a message runs the relevant version of our algorithm based on the target applications. The input to the algorithm consists of the state of the node’s links to its neighbors, as well as the path vector information that the node has received from its neighbors. The resulting multicast tree is then encoded in the message. The intermediate overlay nodes are free of any additional computations or keeping any per-session multicast state information—they only forward the data to their neighbors according to the information embedded in the data.

The multicast tree calculated by the source node can either be encoded in full in the message (with an overhead of $O(N)$), or be encoded as a digest, e.g., using fixed-size Bloom Filters that can significantly reduce the overhead with negligible false positives [48]. In this thesis, we only study the calculation of minimum-delay multicast trees, and the detailed encoding of the tree is outside the scope of our work. Nevertheless, we highlight that the MSDOM algorithms do not need to specify the full tree structure for the intermediate forwarding nodes—they allow the use of Bloom Filter digests. The MMDOM algorithms, on the other hand, require to signal the full tree structure (as in the similar work in [27]), because a forwarding order among the children of each node needs to be specified so that the forecasted maximum delay can be actually met in the network; see $q_{u,T}(v)$ in Eq. (7.1).

The routing table underlying our algorithms is illustrated in Figure 7.3 and consists of a $d_G(s) \times N$ matrix at each node $s$ (based on which the $G_{PV}$ view is created at $s$). An entry $(i, j)$ of this matrix represents the shortest path to overlay node $j$ through the $i$-th neighbor of $s$. Row $i$ of this matrix is maintained over time according to the path vector information that the node receives from the corresponding neighbor. Each path vector entry from this neighbor represents the shortest path of this
neighbor to some overlay node: The path’s hops and hop-by-hop distances (i.e., $O(L_{\max})$). Moreover, a min-heap is maintained on each column of the table, to quickly return the best neighbor for reaching each destination. The complexity of maintaining the routing table is analyzed shortly.

At a high level, the MSDOM-$e$ algorithm gradually adds nodes to the tree according to the total cost of any possible attachment of new nodes to the tree. This total cost includes the delay to the new node as well as the impact on the existing nodes affected by the attachment. The MMDOM-$e$ algorithm repeatedly finds the receiver with the farthest distance (combined link and nodal delay) and attaches it to the tree, while carefully considering the change in all distances caused by attaching each new node.

These two algorithms best suit applications with up to a few hundred nodes which seek to minimize either the total delay in delivering messages or the maximum delay observed across all receivers. The MSDOM/MMDOM-$f$ algorithms take a different approach. They first build a regular shortest-path tree to the given set of receivers. Then, the tree is gradually refined by re-routing to some of the nodes, such that the total or the maximum delay is minimized. These two algorithms suit larger overlays where neither MSDOM/MMDOM-$e$ nor previous works are fast enough. A comparison based on numerical results can be found at the end of Section 7.3.

7.2.2 Detailed Multicast Algorithms

Our multicast algorithms are specified in Figures 7.4 to 7.6. In the MSDOM-$e$ algorithm, we start from the source $s$ and build the multicast tree by incrementally adding nodes according to their cost for the tree. Specifically, for any potential attachment $(u, v)$ to tree $T$ where $u \in T$ and $v \notin T$ (Lines 4 and 6), we find the cost as the increase in the expected total delay to all nodes caused by this attachment (Lines 8 and 9). This delay consists of the expected delay to node $v$ itself, as well as the expected delay to be suffered from by other descendants of $u$ since the degree of $u$ is going to increase (see Eqs. 7.1 and 7.2). Having applied the minimum-cost attachment $(u^*, v^*)$ (Lines 14 and 15), we update the current distance value (array $t$) of all affected nodes (Line 16), i.e., the children and all descendants of node $u^*$. Finally, we clean up the tree (Line 18) by keeping only the paths that end at some receiver node.

The MMDOM-$e$ algorithm repeatedly runs our modified version of the Dijkstra algorithm to find the farthest node from the current tree $T$. This modified shortest-path algorithm (Lines 3 to 14) can start from multiple nodes (see the initialization of $dist[]$ in Line 5), considers the current degree of nodes in the existing tree (Line 5) and expands based on aggregated link and nodal delays given the current tree (Line 11). Having found the farthest node $v^*$ from the tree, $v^*$ as well as its predecessors on the path starting from some node in $T$ (i.e., nodes in list $H$ in Line 17) are added to the tree. Note that after
MSDOM-e and MMDOM-e Algorithms

**MSDOM-e()**

1. $T = BuildEmptyTree(s); t[i = 1, \ldots, N] = \infty; t[s] = 0$
2. while $R \neq \emptyset$ do
3.   $cost[] = 0; prev[] = \emptyset$ // To find the best node to attach to $T$
4.   for $v$ in $V - T$ do
5.     $cost'[\cdot] = \emptyset$ // To find the best attachment point for $v$
6.     for $u$ in $G_Pv$.neighbors($v$) s.t. $u \in T$ do
7.       // Cost of attaching $v$ to $T$ through $u$:
8.       $cost'[u] = t[u] + w(u, v) + (d_T(u) + 2)/2 \times \Delta_u(z)$
9.       $cost'[u] += 1/2 \times \Delta_u(z) \times (g_T(u) - 1)$
10.      $u^* = \argmin(cost')$ // Best attachment point for $v$
11.     if $u^* == NULL$ then continue
12.     $cost[v] = cost'[u^*]$
13.     $prev[v] = u^*$
14.     $v^* = \argmin(cost); u^* = prev[v^*]$ // $v^* \notin T, u^* \in T$
15.     $T$.attach($v^*$ through $u^*$)
16.     $Update_t(T, u^*, t[])$
17.     if $v^* \in R$ then $R$.remove($v^*$)
18.     $CleanUp(T, s)$
19. return $T$

**MMDOM-e()**

1. $T = BuildEmptyTree(s); t[i = 1, \ldots, N] = \infty; t[s] = 0$
2. while $R \neq \emptyset$ do
3.   for $u$ in $V$ do
4.     $prev[u] = -1$
5.     $dist[u] = t[u] + d_T(u) \times \Delta_u(z)$
6.     $S = V$
7.     while $S \neq \emptyset$ do
8.       $u^* = \argmin_{u \in S}(dist[u])$
9.       $S$.remove($u^*$)
10.      for $v$ in $G_Pv$.neighbors($u^*$) s.t. $v \notin T$ do
11.         $d = dist[u^*] + \Delta_u^*(z) + w(u^*, v)$
12.         if $d < dist[v]$ then
13.             $dist[v] = d$
14.             $prev[v] = u^*$
15.         // Attach the farthest node to $T$:
16.         $v^* = \argmax_{v \in R_T}(dist[v])$
17.         $H =$ List containing hops of path to $v^*$ according to $prev[]$
18.         for $v$ from $H$.FirstNodeNotInTree($T$) to $H$.last() do
19.           $u = prev[v]$ // Parent of the to-be-attached node $v$ in $T$
20.           $T$.attach($v$ through $u$)
21.           $t[v] = dist[v]$
22.         if $v \in R$ then $R$.remove($v$)
23. return $T$

† Variables used in the code are described in Table 7.1.

Figure 7.4: MSDOM-e and MMDOM-e algorithms.
this addition, the degree of a number of nodes changes, making the recently calculated shortest-path distances no longer accurate. Thus, the next farthest node is searched for again in the next iteration.

The running time of the MinSum and MinMax algorithms presented so far, as we analyze shortly, is \( O(N^2 D_{PV}) \) where \( D_{PV} \) is the average degree in the path-vector based view of the overlay graph. This running time may not be efficient enough for large overlays, as quantified in the next section. We therefore develop additional algorithms, MxDOM-\( f \), which are optimized for speed.

In these algorithms, which are illustrated in Figure 7.5, we first calculate the regular shortest-path tree from \( s \) to the receivers. This is simply done by merging the shortest paths to the destinations given in the path-vector routing table. This tree is then refined according to the given objective: Minimizing the total delay or the maximum delay.

In the MSDOM-\( f \) algorithm, the tree is refined in a top-down manner from the root downwards. For each node \( u \), we look at each of its children \( v \) and consider routing to it through an alternative route (Lines 5 and 6 of function RefineTree[MinSum]). The alternatives for a node \( v \) are to route to \( v \) through any other of the \( d_G(s) \) neighbors of the root \( s \) than the one currently used to reach \( v \). Given the path vector information available at \( s \), we evaluate the possible alternative routes, and change the current route to \( v \) if necessary (Lines 7 to 10). To refine the route to the children of node \( u \), we first consider those children that have the highest number of receivers in their subtrees (the sort operation in Line 2), since any saving in the delay to those children will likely yield a higher overall saving. Once finished relaxing the degree of node \( u \), the algorithm proceeds with the next level which is refining the subtree of each child of \( u \).

For the MMDOM-\( f \) algorithm, we first note that the maximum delay in the subtree rooted at a node \( u \) can vary much based on the ordering of \( u \)’s children to receive the message (see \( q_{u,T}(v) \) in Eq. 7.1). Thus, we need to obtain an optimal ordering for the children of \( u \). Denoting by \( h_T(v) \) the delay from \( v \) to its farthest descendant, the optimal ordering of \( u \)’s children for minimizing the maximum delay corresponds to sorting the children in descending order of their \( h_T(v) \) values. This is done for all nodes of the tree in function \( T.FixForMinMax \) (Line 1; similarly done later for any affected node in Line 15). In each iteration, the algorithm picks the node with maximum delay (Line 3) and tries to find an alternative shorter route to the node (Lines 5 and 6). If the tree reaches a state where the distance to the farthest node cannot be shortened (Line 8), the algorithm tries to escape the local maxima (Line 9) by re-routing to one of \( v \)’s neighbors (up to 2 hops) such that \( v \) can be re-routed to through a shorter path in the next iteration (a few times in a tree in our experiments). If that is also not possible, the algorithm terminates. Moreover, the total number of refining steps is limited by a bound (MAX,REFININGS in the code). We set this bound to \( N/\log N \) to keep the worst-case running time to \( O(N^2) \) (analyzed
MSDOM-f and MMDOM-f Algorithms

MSDOM_MMDOM_f(mode)†
// mode: MinSum or MinMax.
1. \( T = \text{BuildRegularShortestPathTree}(s) \)
2. \( \text{RefineTree}[\text{mode}](T, s) \)
3. \( \text{return } T \)

RefineTree[MinSum](T, u)
1. if \( T.\text{IsLeaf}(u) \) then \( \text{return} \)
2. \( C[] = T.\text{children}(u) \) sorted in descending order of \( g_T(C[i]) \)
3. for \( v \) in \( C[] \) do
4. \( t'[] = \emptyset \)
5. for \( a \) in \( T.\text{neighbors}(T.\text{root}) \) do
6. \( t'[a] = \text{SavingByRouteChange}(T, v, a) \)
7. \( a^* = \text{argmax}\{t'[a]\} \)
8. if \( t'[a^*] > 0 \) then
9. \( T.\text{DeleteRoute}(v) \) // Detach \( v \) and its subtree
10. \( T.\text{InsertRoute}(v, a^*) \)
11. Sort \( C[] \) again in descending order of \( g_T(C[i]) \)
12. for \( v \) in \( C[] \) do
13. \( \text{RefineTree}(T, v) \)

RefineTree[MinMax](T, u)
1. \( T.\text{FixForMinMax}() \)
2. for \( i = 1 \) to \( \text{MAX}_\text{REFININGS} \) do // Usually breaks earlier
3. \( v = T.\text{FarthestLeaf}() \)
4. \( t'[] = \emptyset \)
5. for \( x \) in \( G_{PV}.\text{neighbors}(v) \) s.t. \( x \in T \) do
6. \( t'[x] = \text{SavingByEdgeChange}(T, v.\text{parent} \rightarrow v, x \rightarrow v) \)
7. \( x^* = \text{argmax}\{t'[x]\} \)
8. if \( t'[x^*] \leq 0 \) then
9. if \( \text{TryEnhancingANeighbor}(T, v) == \text{false} \) then \( \text{break} \)
10. else \( \text{continue} \)
11. \( T.\text{DeleteEdge}(v, v.\text{parent}) \) // Detach \( v \) and its subtree
12. \( T.\text{InsertEdge}(v, x^*) \)
13. while \( v \neq T.\text{root} \) do
14. \( v = v.\text{parent} \)
15. \( v.\text{FixChildrenForMinMax}() \)
16. \( T.\text{FixNodesDS}() \)

† Variables used in the code are described in Table 7.1.

Figure 7.5: MSDOM-f and MMDOM-f algorithms.

shortly). Although, in our experiments the algorithm has terminated much earlier than this bound. To enable efficient reordering of each node’s children and retrieval of the farthest node in the tree, at each node \( v \), we maintain certain information including \( t_T(v) \) and \( h_T(v) \). After each modification in the tree, the information maintained at up to \( O(N) \) nodes may need to be updated. For example, after moving node \( v \) from its old parent \( u \) to a new parent \( u' \), the delay to each descendant of \( v \), the correct ordering of the children of \( u' \) and those of \( u' \)’s ancestors, and accordingly the delay to each descendant of \( u' \) and
**iMSDOM and iMMDOM Algorithms**

**iMSDOM_MMDOM**(mode, $T_0$, $R_0$)

// mode: MinSum or MinMax.
// $T_0$: A previous (old) multicast tree.
// $R_0$: Receiver set for which $T_0$ was built.
1. $T = \text{CopyTree}(T_0)$
2. $\text{RemoveUnwantedNodes}(T, R_0 \setminus R)$
3. $\text{paths} = \text{EmptyVector}(); \text{dist}[1..N] = \infty; \text{next}[1..N] = -1$
4. for $r$ in $R \setminus R_0$ do
5.   $Q = \emptyset$
6.   Reset modified entries of $\text{dist}[1..N]$ to $\infty$; $\text{next}[1..N]$ to $-1$
7.   $Q.\text{insert}(r)$; $\text{dist}[r] = 0$
8.   while $Q \neq \emptyset$ do
9.     $u^* = \text{argmin}_{u \in S}(\text{dist}[u])$
10.    $Q.\text{remove}(u^*)$
11.   for $v$ in neighbors($u^*, G_{PV}$) do
12.     if $v \notin T$ then
13.       $d = \text{dist}[u^*] + \Delta_v(z) + w(v, u^*)$
14.       // Estimate total distance from root to $u^*$ via $v$:
15.       $td = d + \text{PathVectDist}(v) + \delta$
16.       if $d < \text{dist}[v]$ and $td < \text{cost}(path^*)$ then
17.         $\text{dist}[v] = d$
18.         $\text{next}[v] = u^*$
19.         $Q.\text{add}(v)$
20.     else // i.e., $v \in T$
21.       // Stop this branch of search, save candidate path
22.       if $\text{mode} == \text{MinMax}$ then
23.         $\text{cost} = t_T(v) + w(v, u^*) + (d_T(v) + 1) \times \Delta_v(z) + \text{dist}[v]$
24.       else
25.         $\text{cost} = \hat{t}_T(v) + w(v, u^*) + (d_T(v) + 2) / 2 \times \Delta_v(z) + \text{dist}[v] +$
26.         $1/2 \times \Delta_v(z) \times (g_T(v) - 1)$
27.       if $\text{cost} < \text{cost}(path^*)$ then
28.         $\text{path}^* = \text{path from root to } v \text{ (in } T\text{) and then from } v \text{ to } r \text{ according to } \text{next}[\text{]}$
29.         $\text{paths.} \text{add}(\text{path}^*)$ // Best path to $r$
30.     for $p$ in $\text{SortByDescendingCost}(\text{paths})$ do
31.       $T.\text{InsertPath}(p)$
32.   return $T$

† Variables used in the code are described in Table 7.1.

---

Figure 7.6: iMSDOM and iMMDOM algorithms.

$u^*$’s ancestors needs to be updated. Therefore, we simply update the data structures for all nodes of the tree in $O(N)$ time (Line 14).

Finally, the iMSDOM and iMMDOM algorithms can speed up the creation of multicast trees to nearly zero time. These algorithms best suit applications in which the multicast receivers for a source node change gradually, not fundamentally, across consecutive messages, such as in the examples at the beginning of this chapter. Denoting the set of receivers at time $t_0$ by $R_0$, the multicast tree created for $R_0$ by $T_0$ (e.g., using the MxDOM-e algorithm), and the receiver set at time $t$ by $R$, iMSDOM and
iMMDOM takes tree $T_0$ as a base and efficiently updates it according to the difference between $R$ and $R_0$.

First, the nodes that are no longer among the receivers (and the corresponding paths) are removed if they are not used for reaching any other receiver. This is done in Line 2 in Figure 7.6. Then, for adding each new receiver node $r \in R \setminus R_0$, we conduct a search to find the best possible path, and add it to the paths vector (Lines 3 and 26–28). This search starts from $r$ (Line 7), and each path of the search is terminated once reaching some node in the current tree $T$ (the else case in Lines 19–25). In this case, the cost of this path is calculated and $\text{path}^*$ is updated if necessary. Upon visiting each new node $v$ that is not in $T$ and is not visited before through a shorter path, the search continues by recording the distance from $v$ to $r$ in $\text{dist}[v]$ and then visiting $v$’s neighbors later (Lines 12 to 18). To limit the search and keep the algorithm fast, we prune search paths that are unlikely to yield a path comparable to the current $\text{path}^*$. Therefore, an estimated total delay from $s$ to $r$ through $v$ is calculated ($td$ in Line 14), which equals the delay from $v$ to $r$ (variable $d$ in Line 13) plus the delay from $s$ to $v$ ($\text{PathVectDist}_s(v)$ as the link-by-link delay from $s$ to $v$ and $\delta$ as its estimated nodal delay\(^2\) in Line 13).

### 7.2.3 Complexity Analysis

A summary of the memory requirement and running time of our algorithms is as follows. The total memory space for maintaining the required data structures at a node and running our algorithms is on average $O(NDL_{\max})$, where $L_{\max}$ is the maximum hop count in a shortest path (i.e., the hop count of the overlay diameter). Moreover, the time taken by a node to update the routing table, upon receiving a path vector of size $O(NL_{\max})$ from a neighbor, is of $O(N(L_{\max} + \log N))$. The running time of both MSDOM-$e$ and MMDOM-$e$ algorithms is bounded by $O(N^2D_{PV})$. The MSDOM-$f$ algorithm runs in $O(NDL_{\max})$ time. The MMDOM-$f$ algorithm takes $O(N^2)$ time, given that we bound the number of its refining steps (each taking $O(N \log N)$ time) to $N/\log N$ as mentioned earlier; though we note that the refining steps taken by this algorithm in our experiments has been far less than $N/\log N$. The iMxDOM algorithms perform a local search, and while their worst-case running time is $O(N^2D_{PV})$, they terminate much faster (numerical results presented in Section 7.3).

### 7.3 Evaluation

We evaluate the performance of our algorithms on hundreds of overlay networks built on top of two different real-world Internet delay datasets. The overlays are created according to three different overlay

\(^2\)Since $v \not\in T$, the exact value of the total nodal delay from $s$ to $v$ is not known at this time, so we simply use $\delta$ as the average nodal delay across all nodes currently in $T$. This value is only used for pruning unlikely search paths (Lines 14 and 15) and has proven highly effective in our experiments.
Chapter 7. Minimum-Delay Message Distribution

graph models, and range from hundreds to thousands of nodes. Our evaluation setup and the obtained results are presented in this section.

7.3.1 Evaluation Setup

To capture the actual delay between hosts in the Internet, which is a key factor determining the effectiveness of any overlay multicast algorithm, we use the data from two different measurements: The Delay Space Synthesizer (DS\(^2\)) project [96] which provides the measured pairwise delay between about 4000 Internet hosts, and the Meridian project [90] containing the delay between 2500 hosts. We sometimes need to down-sample the set of 4000 (or 2500) nodes to \( N < 4000 \) nodes in our experiments. To ensure having a representative subset, we use a variant of the \( k \)-means clustering algorithm [63] that takes the best \( N \) center points among the original 4000 (or 2500); it minimizes the sum of the squared distance between each original point and its closest center point.

On top of these datasets, we create overlay networks based on three different overlay graph models: Small world [53], random [35] and power law [12]. In small-world networks, links are more likely to exist between nearby nodes than distant nodes [53]. These networks are commonly observed in social and P2P networks [82, 97]; they have also yielded the smallest shortest-path length between nodes in our experiments. We generate these networks by connecting each pair of nodes \((u, v)\) with probability \( \alpha \times \text{distance}(u, v)^{-1} \), where the coefficient \( \alpha \) is set according to the desired average node degree \( D \) in the overlay. On the other hand, random networks are simpler in which all edges of different lengths are treated similarly. Specifically, we generate random networks according to the Erdos-Renyi model [35] where each pair of nodes \((u, v)\) exists with probability \( p \), which we set according to the desired average degree \( D \). In a power-law network (also called a scale-free network), such as the worldwide web, nodes that are popular are more likely to receive new links; the network therefore has a power-law degree distribution [12]. We generate these networks following the Barabasi-Albert model [12], where a new node \( x \) connects to an existing node \( u \) with probability \( \alpha d_G(u) / \sum_{v \in V} d_G(v) \) (\( \alpha \) determining the average degree).

The parameters that we vary in our evaluations include the overlay size \( (N) \), average node degree \( (D) \), number of receivers \( (|R|) \), and node-incurred delays. Specifically, we define \( \Delta \) as the average time that it takes for a typical node to send out each copy of the message in an experiment: \( \Delta_u \) for each node \( u \) captures the combination of node’s bandwidth/load and message size in the experiment. To account for the diversity of nodes’ capabilities in the real world, we also designate 10% of the nodes as supernodes (e.g., behind a campus network) which have 100–1000 times smaller nodal delay than regular
nodes. The value of $\bar{\Delta}$ is assumed the same ($\pm 50\%$) across the regular nodes in an experiment, and as $\bar{\Delta}/10^{U(2,3)} \pm 50\%$ for supernodes; the dynamics of nodal delays are also analyzed shortly. Therefore, supernodes are automatically assigned higher fan-outs by the algorithms (according to their nodal delay).

To get a sense of diverse combinations of node bandwidth and workload as well as message size, the factors comprising the nodal delay, we consider a wide range of values for $\bar{\Delta}$ from 10 ms to 1000 ms in our experiments. For example, on one end, $\Delta = 10$ ms can represent the case where the message is a small 10 Kbits (1.2 KB) packet, and each node has a powerful 10 Mbps upload bandwidth and is forwarding messages of 10 multicast sessions on average; we also analyze the dynamics of nodal delays.

On the other end, consider the delivery of a continuous flow of data where each node would send the data in large chunks to its children in the multicast tree. Assuming TCP transport, to maximize the throughput, the node sends a continuous chunk of up to a full TCP window to its first child, then to the second child, and so on. The message size $z$ can therefore be assumed up to the maximum window size, e.g., 128 KB (1 Mbits) as it is the default value in most Linux distributions. Thus, $\Delta = 1000$ ms can represent the multicast of a continuous flow on the aforementioned network; $\Delta = 1000$ ms can as well represent the multicast of a small 1.2 KB packet through nodes 100 times less powerful.

Given an overlay network of $N$ nodes and a number of receivers $|R|$, in each single run we select a sender and $|R|$ receivers at random from $[1, N]$. We then generate multicast trees using ours as well as previous approaches (described shortly) on the same input: $G_{PV}$ view of the overlay, $s$ and $R$. For each experiment, we repeat this process on several randomly generated overlays, and on 10 different sender/receiver selections on each one of the overlays. The process continues on different overlays until the average result (delay) converges, that is, the deviation of the average result falls and stays below $< 1\%$ for 5 consecutive overlays. A minimum of 10 overlays is also enforced for each experiment even if converged earlier, i.e., 100 runs. We observed that 10–44 and 10–46 overlays were needed respectively for different MinSum and MinMax experiments, with an average of 24 and 29 overlays. Finally, since in the MinSum algorithms a source does not transmit the full ordered tree structure ($\S$ 7.1), we simulate this lack of knowledge in intermediary nodes by shuffling the children of each node in the created tree before evaluating the average delay to the receivers.

7.3.2 Experimental Results

We evaluate the MSDOM algorithms by comparing the achieved average delay with the average delay in the regular shortest-path tree (SPT) for the same overlay and receiver set. We also compare these results with the delay achieved by the algorithm in [64], which builds a MinSum-delay multicast tree
with bounded degrees—the closest work in the literature to our MinSum algorithms, to the best of our knowledge. We run a binary search to find the best degree bounds for this algorithm (each node’s bound is inversely proportional to its forwarding delay), though in our running time measurements presented shortly we only measure the time taken by one run of this algorithm, not the multiple times done in the binary search. This algorithm is labeled as “MLRS” in the following figures (taken from the names of the authors).

We analyze the MMDOM algorithm by comparing the maximum delay in our tree with the maximum delay in the SPT as well as the tree created by the algorithm in [27] for the same overlay and receiver set. As discussed in Section 3.6, the algorithm in [27] is the only previous study on minimizing the joint link and node incurred delays in a multicast tree, to the best of our knowledge. We only consider the heuristic algorithm proposed by the authors, since it outperforms the proposed alternative approximation algorithm (§ 3.6) in both tree-efficiency and running time [27]. We also note that this work addresses only the MinMax version of our problem. This algorithm is labeled as “BLS” in the figures.
Experiments with various overlay/session parameters. We first evaluate the algorithms on overlays of different sizes and present the results in Figure 7.7. We also report the time taken by these algorithms in Figure 7.8. Our experiments are run on a PC with Intel Xeon X5450 3 GHz CPU. The following experiment (Figures 7.7 and 7.8) is conducted on the DS$^2$ dataset, using the small-world overlay model, with $R = N - 1$, $D = N/10$ and $\bar{\Delta} = 100$ ms. The cases with $N = 4000$ are skipped for MLRS, BLS and MxDOM-e algorithms for their running times. Also, we do not report a running time for SPT in Figure 7.8 as it takes a very short time given the up-to-date routing table—only merging the given shortest paths.

It is noteworthy that one might add the running time to the delay achieved by the tree, since a message is not sent out until the tree calculation is done. However, we also note that each calculated tree is typically expected to be used for a number of messages. For instance, back to the example of the dynamic online agent (the source) moving across the area-of-interest of others (the receivers) in a virtual environment, we can realistically expect that the receiver group changes at a quite slower pace (in the order of seconds) than the rate of disseminating messages (several per second). We therefore separately report both times, tree delay and running time. In our last set of experiments in this section we analyze the joint end-to-end delay.

In Figure 7.7, we first observe that the commonly used shortest-path trees, despite being fast to build, suffer from large average and maximum delays as they are unaware of nodal delays. We also observe in Figure 7.7(a) that both of our algorithms outperform the previous related work MLRS [64]. MSDOM-e trees can provide an average delay less than half the MLRS ones, while also being created multiple times faster (note the log-log scale in Figure 7.8). Our fast algorithm MSDOM-f, is several times faster (taking 314 ms for $N = 4000$) while still providing better average delays than MLRS.

Figure 7.9: Tree efficiency for different levels of overlay connectivity ($N = 1000, |R| = 999$).
Similarly, for the MinMax version in Figure 7.7(b), the MMDOM-e algorithm yields the smallest farthest-node delays (< 1.1 s), significantly less (26–66%) than that of the related previous work BLS [27] which is up to 2.8 s. We also observe that the maximum delay achieved by MMDOM-f and BLS are close, nearly overlapping in Figure 7.7(b). The running time of MMDOM-f, on the other hand, is 2 to 3 orders of magnitude smaller—available for scales where none of BLS and MMDOM-e are applicable. Thus, having the MMDOM-e algorithm for overlay sizes of up to hundreds, and MMDOM-f for larger overlays, we can create multicast trees with much smaller MinMax delays (less than half) and/or faster running times (orders of magnitude) than the alternative approaches.

Next, we analyze our algorithms with different levels of network connectivity, defined by different values of the average node degree $D$. Figure 7.9 shows the average and the maximum delay in overlays of size $N = 1000$ nodes; $|R| = 999$, $\bar{\Delta} = 100$ ms, and the overlays are built on the DS$^2$ dataset and using the small-world model. As expected, the regular shortest-path tree that ignores node degrees results in an increasingly high delay as network connectivity increases in Figures 7.9(a) and 7.9(b). This is because shortest paths tend to be formed within very few hops of the source as the overlay gets denser. This yields high node degrees in the multicast tree, and accordingly high delays incurred at overlay nodes. The delays in multicast trees created by our algorithms are several times smaller. In addition, both MSDOM-e and MMDOM-e algorithms achieve a delay often less than half that of the related works (BLS/MLRS). Moreover, MSDOM-f trees provide an average delay between 31% to 46% smaller than MLRS trees, while also being created 2 to 17 times faster. For the MinMax version, similar to the previous experiment, the farthest-node delay in MMDOM-f and BLS trees are tightly close, while the former one is created in 3–13 ms in this experiment and the latter in 3–5 seconds.

In the next experiment, we vary the number of receivers $|R|$; the other configurations remain the
same and $D = N/10$. As plotted in Figure 7.10, the same trend as in the previous experiment is observed among the achieved delay and running time of the algorithms.

Moreover, we analyze the impact of $\bar{\Delta}$, the average nodal delay for each copy of the message, when varying in the range 10–1000 ms as described earlier. For the MinSum version (Figure 7.11(a)), we observe that a similar superiority among the trees made by different algorithms holds as in previous experiments, with roughly the same ratio among them for different $\bar{\Delta}$ values, e.g., MSDOM-e trees have an average delay between 61% to 80% less than MLRS trees. In the MinMax version, the trend is more or less the same as earlier experiments except for small $\bar{\Delta}$ values, given that the dominant factor in the farthest-node delay becomes the longest shortest-path distance (i.e., the diameter) as the nodal delay approaches small values (particularly 10 ms). This effect is not as bold in the MinSum case since it deals with the average shortest-path distance which has been several times smaller than the maximum (the diameter) in our experiments. We also see in Figure 7.11(b) that the MMDOM-e algorithm yields the smallest delays (up to 70% less than BLS), while still running slightly faster than BLS (2x). Furthermore, the MMDOM-f and BLS algorithms once again produce tightly close delays while the running time difference of 2 to 3 orders of magnitude still holds.

**Dynamics of node-incurred delays.** In an actual system, the nodal delays ($\Delta_u(\cdot)$), just like the round-trip delay between nodes (i.e., link delays), are not static values. Although each node $u$ announces its $\Delta_u$ value (short for $\Delta_u(1)$, to be precise) as a representative average over time, e.g., EWMA-averaged, the momentary nodal delay of $u$ at the time of forwarding our message may be considerably different than the announced average. Neither ours nor previous algorithms are specifically designed to capture the uncertainty of these delays. Nevertheless, to have a thorough evaluation study, we analyze the impact of $\Delta_u$ dynamics on the efficiency of the different multicast trees. Therefore, right before evaluating a created
tree, we change the $\Delta_u$ value of each node $u$ to have a random increase or decrease by multiple times: Denoting the variation factor of a nodal delay by $v$ ($v = 1, 2, \ldots$), we change each $\Delta_u$ to a randomly selected value in $[\Delta_u/v, \Delta_u v]$; that is, $\Delta_u$ is multiplied by $e^x$ where $x \sim U(-\ln v, \ln v)$. Figure 7.12 shows the impact of these dynamics. As expected, the average and maximum delay of all trees rise by the increased fluctuation of nodal delays; we should also note that this is partially because the average nodal delay in the network increases by 8–50% for $v = 2, \ldots, 5$. We observe that with varying $\Delta_u$ dynamics, the performance of the algorithms relative to each other remains more or less the same, with MSDOM/MMDOM-e algorithms always yielding trees with lowest delays.

Datasets/Network models. We evaluate our algorithms on both datasets DS$^2$ and Meridian, using all the three network models of small world, random, and power law. The results, including tree delays and running times, are presented in Figure 7.13. The running times for SPT are omitted as they are negligible—only merge the shortest paths given in the routing table. These experiments are conducted on overlays of size $N = 1000$ with $R = 999$, $D = 100$ and $\bar{\Delta} = 100$ ms. The experiments for each of the two datasets, represented by the first two sets of bars in Figures 7.13(a) and 7.13(b), are conducted on all the three overlay models and the results are averaged. Similarly, the experiments for each overlay model, represented by the last three sets of bars in each figure, are run on both datasets and the average is plotted. We observe in these figures that the different algorithms perform more or less the same as in the previous experiments: MSDOM/MMDOM-e algorithms produce the most delay-efficient trees (73–80% smaller delays than MLRS and 52–60% than BLS) while also running slightly faster than the previous approaches; note the running times written on top of the bars. MSDOM-f also has better tree efficiency (24–50%) and running time (6–8 times) than MLRS. MMDOM-f runs in 10–14 ms for the same inputs on which MMDOM-e and BLS algorithms have taken seconds, while still creating trees.

Figure 7.12: Dynamics of nodal delays ($\Delta_u$).
Chapter 7. Minimum-Delay Message Distribution

(a) MinSum-delay trees.

(b) MinMax-delay trees.

Figure 7.13: The performance (tree-efficiency as well as running time) of the algorithms on different datasets and overlay models. This and the following figures are best viewed in color.

with reasonable delay-efficiency; compare it to SPT which is the only applicable alternative based on running times.

Algorithms for tree adaptation. In this experiment we evaluate the efficiency of multicast trees produced by our tree adaptation algorithms, iMSDOM and iMMDOM. In the next experiment we show how to best employ these algorithms on top of MxDOM-e/f. In each run of this experiment, we multicast messages from the same source node $s$ to a series of partially different receiver groups $R_1, \ldots, R_{10}$: $R_1$ is randomly generated as before, and each $R_{i+1}$ is obtained by adding and removing up to $MAX\_CHURN$ nodes to/from $R_i$—both numbers uniformly selected in $[1, MAX\_CHURN]$. This is to simulate the churn of nodes listening to a source node $s$ over time. Each of these 10-session experiments is repeated 100+
Chapter 7. Minimum-Delay Message Distribution

(a) MinSum-delay trees.

(b) MinMax-delay trees.

Figure 7.14: Performance of our tree adaptation algorithms on overlays of different sizes.

(a) MinSum-delay trees.

(b) MinMax-delay trees.

Figure 7.15: Performance of our tree adaptation algorithms with different receiver churn rates.

times as described earlier.

Figure 7.14 plots the results of this experiment on overlays of different sizes, with $D = N/10$, $\Delta = 100$ ms, $|R_i| = N/2$ and $MAX\_CHURN = 10$, i.e., up to 10 nodes are added and up to 10 nodes are removed between $R_i$ and $R_{i+1}$. Similarly, Figure 7.15 shows the sensitivity of our algorithms to the rate of churn in the receiver set, i.e., varying $MAX\_CHURN$ from 5 to 50, with $N = 1000$. In the experiments in Figures 7.14 and 7.15, the MxDOM-e and MLRS/BLS algorithms are skipped for $N = 4000$ for their running times. The aqua and brown bars which are labeled as “iMSDOM-f” and “iMSDOM-e” (similarly for MMDOM) represent the iMSDOM algorithm when using MSDOM-f and MSDOM-e tree (for $R_1$) as a base and then adapting it for $R_2, \ldots, R_{10}$. The running times reported in Figure 7.14 are those of iMSDOM only (all under 1 ms), where the base tree for $R_1$ is given. We show in the next experiment how to avoid the delay for obtaining the base tree.
Figures 7.14 and 7.15 confirm the efficiency of our tree adaptation algorithms, which match the minimized delay of the given base tree in almost zero time (<1 ms). In particular, \(i\)MMDOM-\(f\) produces trees with a maximum delay no higher than MMDOM-\(f\) trees in Figure 7.14; in some cases it even slightly enhances MMDOM-\(f\) trees by \(\sim 1\%\). More importantly, the delay achieved by \(i\)MMDOM-\(e\), which runs in nearly no time (<1 ms) compared to the most delay-efficient algorithm MMDOM-\(e\), is only 3% higher in Figure 7.14(b), and up to 4% higher in the extreme case of \(MAX\_CHURN = 50\) in Figure 7.15(b), i.e., the receiver group is fundamentally changed as going from \(R_1\) to \(R_{10}\).

Unlike the MinMax version where the maximum delay is sensitive to the addition of some nodes to the tree (and to the corresponding degree increments)—or even the added node itself may become the farthest node—in the MinSum version the average delay is not significantly impacted by these changes. This is observed in Figures 7.14(a) and 7.15(a) where \(i\)MSDOM-\(e\) and MSDOM-\(e\) trees have nearly the same average delays. Also, \(i\)MSDOM-\(f\) trees have even up to 8% smaller delays than MSDOM-\(f\), as they specifically take advantage of the similarity of the base and the current tree whereas the pruning-based from-scratch algorithm MSDOM-\(f\) does not; hence this does not indicate the superiority of \(i\)MSDOM-\(f\) in the general case (\(R_{i+1}\) independent of \(R_i\)).

In summary, in the common case of multicasting to a dynamic receiver group that changes gradually over time, our tree adaptation algorithms should be employed which can produce trees in sub-millisecond time even for large overlays, yet with delays close to that of the most efficient algorithms MSDOM-\(e\) and MMDOM-\(e\) which can take up to several seconds for large overlays. Nevertheless we note that our tree adaptation algorithms operate on a given base tree, which should be prepared as fast as just a few milliseconds so we can fully realize the benefits of \(i\)MxDOM algorithms. The next experiment demonstrates this process.

**Real-time multicast with the MxDOM algorithm set.** Consider a source node continuously distributing messages to a highly dynamic group of receivers over time \(R_1, \ldots\). While in practice the rate of the stream of messages is usually faster than the rate of updates to \(R\) (hence reusing each calculated multicast tree for multiple messages), let us consider the extreme example of having an updated receiver set for every single message, i.e., a new multicast tree is required for every message. These messages are generated at a rate of 10/sec in this experiment.

Although the MxDOM-\(e\) algorithms produce the most delay-efficient trees, they are not fast enough for large or medium scale overlays and cannot catch up with generating fresh trees at this fast pace. \(i\)MxDOM can address this problem by adapting a previously created MxDOM-\(e\) tree for each updated receiver set within just a milliseconds, but the efficiency of trees will degrade as the receiver set gets less and less similar to the one corresponding to the base tree. Periodically renewing the base tree using
Figure 7.16: Real-time multicast of a stream of messages with the MxDOM algorithm suite.

MxDOM-e, on the other hand, will produce noticeable hiccups in the stream of messages.

We solve this problem by running our algorithms concurrently in multiple threads. The first thread keeps creating MxDOM-e trees, although at a slower pace than that of receiver set updates (10 times per second). Once done with calculating a tree for \( R_i \), the current receiver set may have been updated several times and reached \( R_{i+j} \). This thread immediately moves on to creating a tree for \( R_{i+j} \) and misses the receiver sets in between. It therefore produces some renewed, though belated, base trees for iMxDOM. The second thread is triggered on each update in the receiver set. It takes the latest MxDOM-e tree as base and quickly updates it for the current receiver set using iMxDOM. For comparison, we also run a third thread that keeps running the MxDOM-f algorithm on the current receiver set. This algorithm is fast enough to keep up with making trees but not as delay-efficient as the combination of MxDOM-e and iMxDOM. BLS and MLRS algorithms are not applicable for this experiment; even for creating just the base tree, they are already outperformed by MxDOM-e in both speed and efficiency. We also create SPT trees in this experiment as a baseline. In all cases, we add the delay achieved by the multicast tree with the time it takes to prepare the corresponding tree, and report the total end-to-end delay.

Figure 7.16(b) shows the results of this experiment for a 10-minute run, where a source node multicasts 10 messages per second, the receiver set gets updated for each message with \( MAX\_CHURN = 5 \) (i.e., up to 5 addition and 5 removals from \( R_i \) to \( R_{i+1} \); up to 50 per second), in a scenario with \( N = 1000, |R_1| = 500, D = 100 \) and \( \bar{\Delta} = 100 \) ms. The significant benefit of employing iMxDOM on top of MxDOM-e is clearly demonstrated in Figure 7.16(b): 54%/63% smaller end-to-end delay compared to MxDOM-f in the MinSum/MinMax version, and 72%/77% compared to SPT; as reviewed earlier, SPT and MxDOM-f are the only applicable (fast-enough) algorithms for this scenario, to the best of our knowledge.
### 7.2 Summary of the MxDOM algorithm suite.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSDOM / MMDOM</td>
<td>Algorithm names standing for: MinSum and MinMax Overlay Multicast.</td>
</tr>
<tr>
<td>MSDOM-e / MMDOM-e</td>
<td>Our algorithms with highest delay-efficiency. Outperform previous approaches in both tree efficiency and running time. Best for overlays of up to a few hundred nodes.</td>
</tr>
<tr>
<td>MSDOM-f / MMDOM-f</td>
<td>Our algorithms optimized to be fast. Best for overlays of up to a few thousand nodes, making trees with ~50% smaller delay than SPT, the only applicable alternative for this scale.</td>
</tr>
<tr>
<td>iMSDOM / iMSDOM</td>
<td>Our algorithms to incrementally build multicast trees w.r.t. recent changes in the receiver group, making delay-efficient trees in nearly zero time. Best for multicasting a stream of messages to a group of receivers churning continuously.</td>
</tr>
</tbody>
</table>

### 7.4 Summary

This chapter aimed at delivering messages from a source host to a group of receivers with minimum end-to-end delay, including the total of delays incurred in overlay links and at high-degree forwarding nodes. A suite of algorithms was presented which addresses two versions of the problem, minimum-average (i.e., minimum-sum) and minimum-maximum delay delivery, and for each case it supports a diverse range of application requirements. A summary of the algorithms is given in Table 7.2. In this algorithm suite, the MxDOM-e algorithms outperform existing approaches in both tree efficiency and running time, and are the best choice for overlays of up to a few hundred nodes. For larger overlays, where previous approaches as well as MxDOM-e are not feasible, MxDOM-f algorithms are suggested which can run fast (40–90 ms for $N = 2000$ and 160–310 ms for $N = 4000$ in our experiments in Figure 7.8) while producing trees with reasonable delay efficiencies: 40–60% smaller delays than SPT trees which are the only applicable alternative. Finally, iMxDOM algorithms can update multicast trees, instead of rebuilding them from scratch for each message, for multicasting a stream of messages to a group of receivers with continuous churn. iMxDOM algorithms run in sub-millisecond time and produce trees with comparable delay efficiency to our other algorithms. The evaluation study also showed how our algorithms can be employed together to yield the smallest end-to-end delay in such (churned) streaming scenarios.
Chapter 8

Conclusions

In this thesis, we have developed multiple solutions for efficient, planet-scale video content distribution as well as algorithms for distributing delay-sensitive messages in large distributed systems. We studied the content caching problem in video CDNs and its novel requirements, and we developed an LRU-based solution (xLRU Cache) as well as a more efficient algorithm specifically for ingress-constrained servers (Cafe Cache). We also studied the offline caching problem assuming knowledge of future requests. We designed a greedy algorithm based on this knowledge (Psychic Cache) which heuristically estimates the maximum caching efficiency we can expect from online algorithms, and an LP-relaxed optimal algorithm for limited scales (Optimal Cache). We have shown using real video CDN traces that while xLRU can be good enough where cache ingress is not expensive, Cafe Cache performs with over 12% higher caching efficiency for the common case of servers with ingress constraints, achieving an efficiency relatively close to the offline Psychic algorithm. Moreover, we have conducted a detailed analysis of the correlation of video popularities worldwide based on real CDN workload data. We found that video popularities are strongly correlated within a metro or even a country, even with lingual diversity (>0.5 correlation). We also found that the workloads of nearby countries may not be correlated, and correlated countries may not be nearby since language is the main driving factor for inter-country workload correlations. Though, even within the realm of one language, interests may be more local (Spanish) or less (English). Moreover, we performed a detailed analysis on the benefits and the scalability of distributed cache coordination across a global video CDN. We have built a practical cache coordination mechanism for real-world CDNs and showed that exchanging video popularity data as infrequently as every 10 minutes between servers can achieve nearly all the traffic saving that a frequent 10-second exchange gets (up to hundred of Gbps in costly traffic CDN-wide; petabytes per day), while it reduces the coordination overhead to be
negligible. Moreover, our results have shown a close relationship between the correlation of workloads and the profitability of cache coordination. The results also showed that arbitrary coordination can actually worsen the caching performance compared to just leaving the caches non-coordinated. We have analyzed the extent to which cache coordination can scale, which guides the formation of coordination groups throughout a CDN. We found that up to country-wide server coordination is beneficial even in countries with lingual diversity, while coordination between countries is generally ineffective or harmful given the associated overheads. The several intuitive and non-intuitive findings we present through these analyses, each quantified and validated by real-world data, can lead the design of different content distribution components such as coordinated caching, traffic mapping and server provisioning. In addition, we studied the provisioning of a cluster of video CDN servers. We designed an adaptive ingress control scheme, specifically for periods of peak load and excess demand, which reduced a server cluster’s peak upstream traffic by up to 17% and it’s peak-time disk write load to close to 0, hence reducing the network bandwidth and disk size to provision. We also analyzed the interaction of cache layers inside a server, such as SSD and RAM above the disks. We designed an optimization framework to find the right server count, server configuration and peering bandwidth to deploy at a given location. Finally, we studied the generic problem of delivering messages to a group of receivers on an overlay network with minimum end-to-end delay, including the delay on overlay links and on high-degree nodes in the multicast tree. We proved the hardness and the inapproximability of the problem to within any reasonable approximation ratio. We designed a suite of algorithms for this problem: for each of the two minimum-sum and minimum-maximum delay cases, we designed a delay-efficient algorithm, a scale-efficient one that is orders of magnitude faster, and a tree adaptation algorithm to update a previous tree in nearly zero time. The collection of these algorithms supports a wide range of overlays sizes, real-time requirements and session characteristics (rate of messages and churn). Extensive experimentation on two real-world Internet delay dataset showed that these algorithms can achieve significantly lower delays (up to 60%) and smaller running times (up to 3 orders of magnitude) than previous approaches.

The limitations of our work and the promising research problems for the next steps are listed in the following.

**CDN-wide optimality with Cafe cache.** In Chapter 4 we have focused on an optimized caching algorithm for individual servers based on the given ingress-to-redirect preference $\alpha_{F2R}$. As discussed in Section 4.2.1, the primary target of these algorithms is to relieve ingress-constrained servers; see the same section for generic assignment of $\alpha_{F2R}$ values in a CDN. Nevertheless, Cafe cache with defined operating points through $\alpha_{F2R}$ (Figure 4.5) provides the necessary building block for managing traffic between any group of constrained/non-constrained servers. This can be done via tuning of $\alpha_{F2R}$ for servers that
are interrelated according to the CDN topology, e.g., reducing ingress on one increases redirected traffic landing on another. Capturing different inter-server relations (based on real CDN topology data or a related model) and modeling the interactions based on Café cache such that the global CDN can be optimized is an interesting problem for future research.

**Optimal cache.** We have not addressed the problem of optimal offline caching thoroughly. An exact optimal solution for actual scale, whether the proposed IP formulation in this thesis or a customized algorithm, and/or analysis of the tightness of the LP-relaxed version proposed in this thesis can be a beneficial future study.

**Content provider preferences.** General CDNs such as Akamai distribute the content of different content providers. The caching algorithms we designed for a CDN treat all content equally. In some cases, however, different providers may prefer different customizations of the delivery service, such as in terms of startup latency or backend access. Capturing such requirements in the content management algorithms is a useful problem for further study.

**Information for better caching.** The information underlying our caching algorithms is the request streams seen by the CDN servers. Additional types of information either from the requests themselves or from side channels may be used to potentially improve caching decisions further. For example, one can take into account the requests for other versions of a video (encodings, resolutions) or information on how long/short into a video is watched as an indication for the video being more/less popular. One can also incorporate side information such as “Related videos” provided by most content providers. While our anonymized workload traces do not provide such information, this may be potentially fruitful information for caching. For example, a server could update the popularity information for video $v$ upon serving a related video of $v$—or serving another version of $v$ or if $v$ is watched in long sessions—similarly to how it would update the information for $v$ upon playback of $v$ at a neighbor server. In a more radical approach, one can feed these and any other potential feature to a learning model and analyze their relevance for better caching.

**Automated cache coordination configuration.** Our work in Chapter 5 provides guidelines for static configuration of cache coordination clusters throughout the CDN. In the next step, one can develop an automated method for configuring cache coordinations dynamically by sampling the workloads landing on each serving site, analyzing their correlations and possibly learning the profitability of coordination via launching controlled trials. This is a specially useful technique for the possible discrepancy between the user-to-server mapping criteria of a CDN and the users’ geography, which is not uncommon for large serving sites; that is, we cannot define a bounded geographic region for the users of these sites.

**The aggregated saving of cache coordination.** We quantified the savings of cache coordination
at different scaling steps, though not the exact \textit{compound} saving CDN-wide. As discussed, this requires concurrent simulation of a remarkably large number of server locations with per-server traces in a reasonable size (not significantly sampled)—a practice beyond our current means. A useful followup to the current work is to design methods and/or models to quantify the compound savings.

**Optimized cache coordination.** The analyses in Chapter 5 provide the basis for a fine-tuned, production-level coordinated caching system; they show that coordination is worth the effort. Further optimizations that can be done in the next steps include more involved exchange of popularity data (e.g., using Bloom Filters) and more complex state syncing (e.g., differentiating videos reported by most or just a few neighbors); whether they can achieve a gain worth their complexity.

**Analysis of coordinated caching with per-server ingress-vs-redirect configurations.** While the cache coordination mechanism we designed is orthogonal to the underlying caching algorithm, our cache coordination experiments are based on the xLRU algorithm with $\alpha_{\text{F2R}} = 1$ as discussed in Section 5.1. An interesting future study is to analyze the possible effect of per-server ingress-vs-redirect configurations (specifically via Cafe cache) on coordination. Note that the information exchanged between servers is the same with or without such configurations. The effect on the servers’ ingress and redirected traffic across the CDN, however, may not be. Also note that realistic analysis of such effect is a CDN-specific task as to the servers’ location in the CDN topology and the assigned traffic volumes and capacities in the CDN (beyond our available data).

**Reliability and availability of the message distribution service.** Our multicast algorithms rely on the routing information exchanged between overlay nodes, including the shortest paths and nodal delays ($\Delta_u$). While being widely used in almost every routing algorithm, this mechanism still takes a short while for notifying a change to all nodes, such as a failed link or an overloaded node. To achieve full robustness and not losing the availability of the minimum-delay routing service even for a short while, we can improve our algorithms to a multi-path version, i.e., sending each message over two paths that minimize the expected delay to the receivers (considering the joint link and nodal delay) while collectively pushing the chance of successful delivery close to 100%.

**Cloud-based/assisted message distribution.** Our minimum-delay multicast algorithms can take best advantage of powerful nodes in the network by automatically assigning higher/lower fanout to each node based on its upload bandwidth. A useful extension to this work is to capture the heterogeneous connections that one node or a group of nodes may have to others. For instance, a tenant server in a datacenter has practically unlimited bandwidth to its neighboring servers while the collection of these neighbors may have a much tighter bandwidth limit to others, e.g., to contain the (peak or 90 %ile usage based) bandwidth cost below a cap.
Bibliography


