Data-Driven Management of CDN Performance

Mojgan Ghasemi

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Electrical Engineering
Adviser: Professor Jennifer Rexford

November 2017

Abstract

Content Distribution Networks (CDNs) carry most of the web content, with the goals of offering good performance to users at a low cost. In this thesis, we introduce measurement and analysis techniques to help CDNs balance these goals.

First, we allocate resources efficiently across the distributed edge servers by jointly minimizing network latency and cache misses. We propose a unified framework for CDNs to jointly solve the problems of placement, mapping, and disk allocation, while including the impact of cache misses. We evaluate our methods using request logs from a commercial CDN. We show that including the impact of cache misses in the post-mapping disk allocation enhances the performance significantly for a small increase in the cost. Still, there are other sources of performance problems on the end-to-end (e2e) path of the servers to clients, which takes us to the next part.

Second, to detect and diagnose performance problems that cause poor experience for users, we propose a fine-grained instrumentation of the e2e path. In particular, we focus on the video delivery path because video is now the dominant application of the Internet. We deploy our instrumentation and diagnosis methods in a commercial content provider, enabling us to join the server-side, TCP statistics, and client-side measurements for the first time, and characterize the performance problems in a large set of videos. We uncover a wide range of problems, some of which were unknown before, and can only be discovered by an e2e instrumentation. While capable of diagnosing these problems, our tool is limited in how frequently it measures network. To remedy this, we propose our final solution.

Finally, we dive deeper into diagnosing network problems by monitoring TCP connections directly in the network devices. Our tool can pinpoint if the performance of a TCP connection is hindered by the sender, receiver, or network. We deploy emerging programmable edge devices to implement our monitoring and diagnosis logic directly in the data plane, which runs at line-rate, without cooperation from

servers. We infer fine-grained TCP metrics from the edge device (e.g, NIC), without imposing storage or monitoring overhead on the servers.

Acknowledgements

I would like to thank my wonderful advisor, Professor Jennifer Rexford, for being the most amazing advisor I could ever ask for. Jen supported my transition into the field of computer networks, and I was extremely lucky to explore this field under her guidance and encouragements. She encouraged me to do internships, and taught me many lessons in all endeavors including doing research in industry. She taught me how to find problems with real-world impact and how to solve them. She taught me how to present my ideas to a wide range of audiences. I consider Jen my mentor in life, not just my advisor in graduate school.

I am extremely fortunate to have worked with Professor Theophilus Benson. Theo guided me in my first project in graduate school when I was new in the computer networking field, and he was very supportive and patient with me. Theo helped me connect with great researchers in both industry and academia, and was a wonderful host to me while I visited Duke university.

I am grateful to have had the opportunity to work closely with Professor Bruce Maggs. I have immensely enjoyed doing research with Bruce. He taught me how to extend a smaller problem to a general system and continued exploring new ideas and brainstorming with me all throughout the development of the CAM project. He connected me with people working on similar problems and helped me broaden my network.

I am grateful to have had the opportunity of doing research in industry. I thank Partha Kanuparthy for being my mentor in Yahoo Research. I thank Jim Wyllie and Karim Abdel Magid Mattar Shaban for mentoring me during my internship at Akamai.

I would like to thank the members of my dissertation committee, Professor Prateek Mittal, Professor David Wentzlaff, Professor Nick Feamster, and Professor Theophilus Benson for serving on my committee and providing feedback on this dissertation.

I also would thank the members of my research group, Cabernet, including current and former students and post-docs: Srinivas Narayana, Praveen Naga Katta, Nanxi Kang, Xin Jin, Peng Sun, Jennifer Gossels, Mina Tahmasbi, Rob Harrison, Robert MacDavid, Ori Rottenstreich, and Ronaldo Ferreira.

My collaborations in industry benefited from many researchers and engineers. I thank the folks in the Yahoo Research and the Yahoo Video Platforms for helping us with instrumentation, answering our questions, and hearing our performance optimization observations. I thank the members of the Media Service Performance at Akamai, specially the Foundation team, for patiently answering our questions and for their constructive feedback and helpful conversations about our ideas.

Finally, I would like to thank my family for their endless love and support. I thank my parents and my sister Nafiseh for always being there for me and teaching me how to be strong. Above all, I thank my wonderful husband, Arthur, who helped me the most during my time in Princeton, and without his love and support I could not have done this. Thank you for your love and support, and for all the cappuccinos. I dedicate this dissertation to you.

To my husband Arthur,

My parents Mohammad and Fariba,

And my sister Nafiseh

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Content Distribution Networks (CDNs) deliver a wide range of content including web objects, video, e-commerce applications, and social networks to users. In the early days of the world wide web, the content was served through the provider's own servers. The client would use the DNS to get the IP address of the provider's server, and then created a TCP connection and requested the content via HTTP (Figure 1.1). However, there were some challenges in serving the content this way. First, performance was not ideal because the server could be located far away from the clients, causing high latency. There was also an availability problem if a server went down. Additionally, there were scalability problems if many requests arrived for a popular object, beyond the load that the server could handle. Finally, there were security concerns such as Denial of Service (DOS) attacks, where a flood of superfluous requests was sent to a target server and would overload the server, making it unavailable to its intended users. A single-server service is particularly an easier target to take down and causes outage faster.

These challenges were the driving factors for content distribution networks or CDNs. A CDN is a set of distributed caching servers across the world, located near

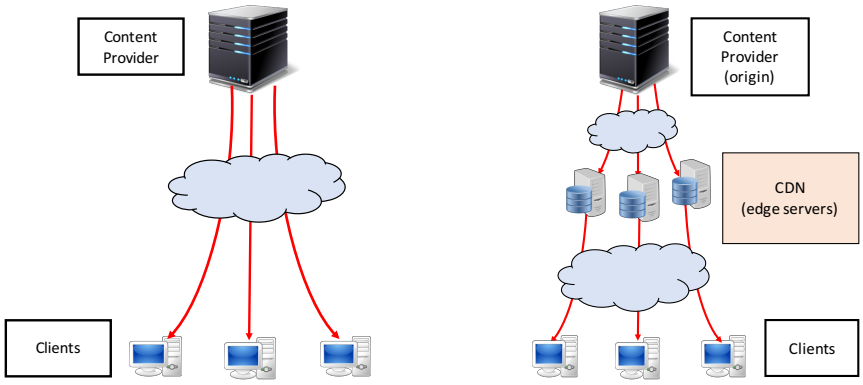


Figure 1.1: How content was served before (left) and after CDNs (right).

clients, to serve the content with higher performance (e.g., lower latency) to the clients. A few example CDNs include Akamai, Limelight, and CloudFare.

When a client requests the content, the request is directed to one of CDN's caching servers across the globe using the DNS protocol. A key component of CDNs is the *global load balancer* (GLB), that chooses the proximal edge server per client's request [29]. If the content is available on the disk at that edge server (i.e., cache hit), the edge server sends it to the client. Otherwise, if the content is not available on the local cache (i.e., cache miss), the server first fetches the content from the origin server before serving it to the client, while also possibly updating the local cache with new content. Note that often an edge server consults other CDN servers (e.g., peers) before requesting from origin servers; however, for simplicity, we have omitted the caching hierarchy in this figure.

The use of CDNs' edge servers resolves the previously mentioned challenges: Performance is enhanced because objects are served from a nearby replica. Availability is enhanced because of the replication of the content. Scalability improves because many servers can serve the content and spread the load. Finally, the security of origin server increases because the DNS protocol will direct the traffic to CDNs' edge

servers, making it harder for the adversaries to overload all of the CDN servers as compared to overloading a single origin server, protecting the origin machines [1].

Today most of the web content is served through CDNs. In particular, Akamai alone served 15-30% of all web traffic in 2016, from over 170,000 servers deployed in over 102 countries and 1300 ISPs around the world [67, 29].

The goal of a CDN is to serve content—usually from a content provider (CP)—to end-users with high *performance* for clients (e.g., serving requests with low latency or high throughput), and at a reasonable *cost* for the CDN. The cost comes from consuming resources such as bandwidth or disk capacity. To achieve this goal, CDNs deploy the edge servers in hundreds of data-centers around the globe; the widespread edge servers ensure that the client's request will be served by a "proximal" edge server. Striking a balance between these two competing goals is often a problem that CDNs face. We will show examples demonstrating this trade-off in Chapter 2. Our goal in this thesis is to help CDN operators manage the performance of the platform at scale; to do this, we must first understand what factors can hurt the performance of the users of a CDN.

1.1 Sources of Poor Performance

There are many sources of poor performance for clients being served by a CDN server:

Far Away Replicas: The GLB often uses the network performance data (e.g., loss and latency) as well as the real-time data from servers (e.g., load and liveness of servers) [29] to map a client's request to an edge server that offers *the best performance at a reasonable cost*. If the chosen edge server is too far from clients (i.e., not proximal), the client experiences high latency, which results in a poor quality of experience for users (i.e., long wait time before the video starts). In addition to poor

[1] An adversary can still attack the target machines if their IPs are known without DNS.

mapping, this issue can also be caused if the CDN does not enough have replicas that are geographically distributed. i.e., even though the client is mapped to the closest server, that server is still far from the clients.

Performance of the Edge Server: Once a request arrives at the edge server, the server begins processing it. For every request, the CDN edge servers need to inspect the local cache including the memory, disk, and finally their peers or origin servers to first find the content and then transmit it to the clients. High latency in serving a client's request may be caused by poor edge server performance. We show instances of such problems from real datasets in Chapter 3. For example, we found asynchronous timers in Apache Traffic Server (ATS)—an open source HTTP proxy—cause extra latency in serving the content from the disk. This issue was impacting a significant portion of requests served by this CDN.

Network: Network problems can manifest themselves in the form of increased packet loss, reordering, high latency, high variation in latency, and low throughput. Each can be persistent or transient (e.g., temporary spike in latency caused by congestion). Such network problems on the path of the client to CDN edge server could impair the performance of users. In Chapter 3 Section 3.4.2, we will discuss network problems in more detail.

Client's Machine: Most Internet video providers split a vidoe into fixed-length pieces, called "chunks". The video player on the client-side makes an HTTP request to fetch each chunk in the appropriate bitrate. We divide the client's machine in two parts:

1. Download Stack: the path that downloaded chunks take before arriving at the application, for example the video player. The path includes the Network Interface Card (NIC), Operating System (OS), and the browser.

2. Rendering Stack: the extra processing performed on video chunks after arriving at the video player, including the de-muxing of audio from video, decoding, and rendering frames on the screen.

We found that a client's poor performance may be caused by the client's bad download or rendering stack. We saw instances where video chunks get buffered in client's download stack and cause video re-buffering. We also found out that some combinations of OS and browsers (e.g., Yandex or Safari on Windows) are more likely to have persistent download stack problems. Recognizing the lasting effect of client's machine on users' quality of experience (QoE) is extremely important for content providers and CDNs. See Chapter 3 for more details.

Cache Misses and Latency To Origin: When content does not exist on the edge server's local disk, it needs to be fetched from the peers or the origin server. This process may itself create a chain of requests if the peer also does not have a copy of the content. Often the origin servers are located far from the edge servers, which increases the latency in serving content significantly because the requests travel a wide area network (WAN). In our study (Section 3) we show that cache misses impact the server latency significantly: the median server latency among chunks experiencing a cache miss is 40 times higher, while the average and 95^{th} percentile is tenfold.

In this section, we detailed the sources of poor performance that cause a CDN server to answer a client's request with high latency or serve the content with low throughput. To detect such problems and manage CDN performance at scale we need tools and techniques that each focus on relevant parts, including optimizing the platform, diagnosing problems along the end-to-end path, and diagnosing finer-grained network problems. We will explain these techniques in the next section.

1.2 Techniques For Managing CDN Performance

Following the key sources of performance problems in CDNs, we propose our scalable performance management and diagnosis system, which consists of three parts, as shown in Figure 1.2. In this management system, CDN operators can first configure the CDN platform to allocate its resources efficiently (Chapter 2). However, this efficient allocation of resources is not enough, as we showed in the previous section; performance problems can arise in a multitude of locations. Thus, the CDN operators also need to quickly detect and diagnose problems when users have poor quality of experience (QoE). To this end, we propose an end-to-end instrumentation methodology per-HTTP transaction (e.g., a chunk in video) that collects measurement data from both sides (client and server) in addition to snapshotting TCP metrics. By joining this dataset we construct performance metrics at essential milestones on the end-to-end path to diagnose performance problems (Chapter 3). While the per-chunk instrumentation can uncover a wide range of problems across the delivery path, instrumenting the CDN servers to collect fine-grained TCP measurement with higher frequency is prohibitive because it consumes resources at the edge server and imposes a storage overhead. To deal with this limitation, we present a tool to allow CDN operators to dive deeper into the network problems and diagnose TCP problems more efficiently directly in the network elements (e.g., network interface card or switch) (Chapter 4).

It is important to note that "measurement" is the key in achieving our goals in the three systems, and is the common theme in all three projects. These tools rely on collecting and analyzing large datasets from the real world systems. We have built prototypes for all three parts and evaluated Diva and CAM in real-world deployments.

CAM: Cache-Aware Mapping. CAM is a framework to allocate resources efficiently across the CDN platform to strike a balance between performance and cost.

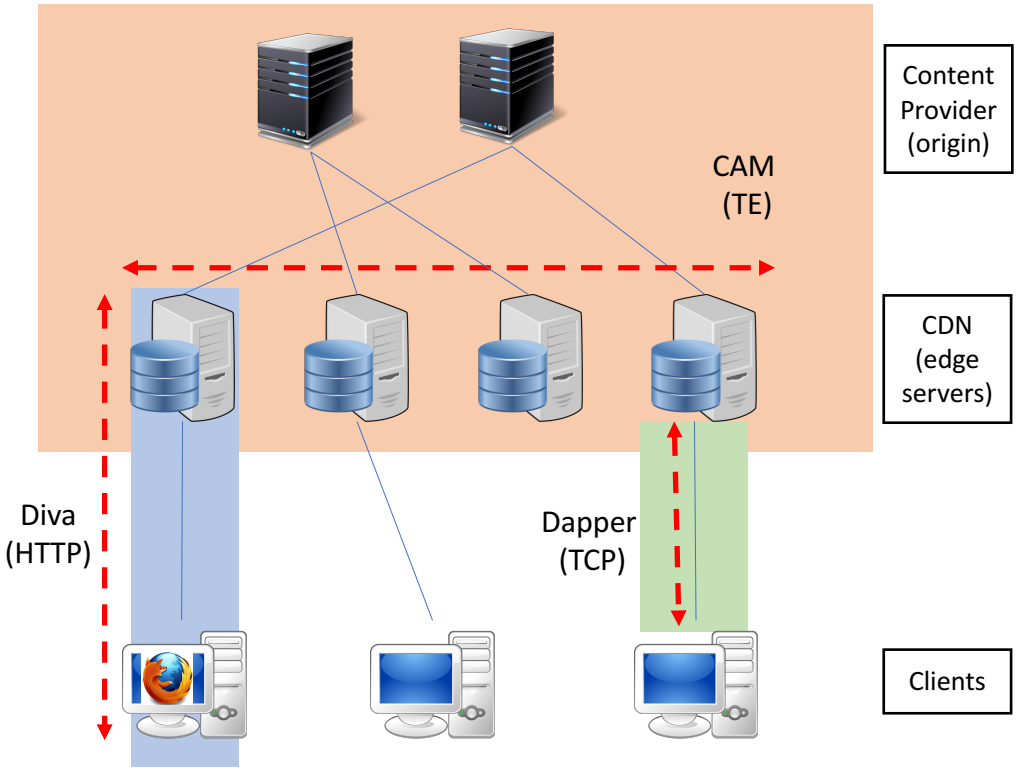


Figure 1.2: Performance management and diagnosis tools: the scope of the three tools

Cache misses have a negative impact on users (e.g., increased rebuffering on video clients), CDN operators (increased costs of operation), and content providers (increased load on origin servers); however, the impact of cache misses has not been explicitly and modeled in a unified framework for CDNs. CDNs optimize content placement (i.e., which edge servers should be allowed to serve the requests of a content provider), request mapping (i.e., which server should serve each request), and disk partitioning (how should the shared cache be managed among several customers). In CAM, we propose a unified performance model that captures these three resource allocation problems, while including the impact of cache misses. We avoid making simplifying assumptions about the popularity distributions of content. While our model focuses on the client's performance, it can easily include cost (to the CDN) and fairness (across content providers) as constraints. Since the joint optimization is a non-linear and non-convex problem, we solve a simpler variation of it by only

focusing on one of these problems. We show that even without changing the current mapping or placement, by only managing the disk efficiently, we can enhance the performance significantly (17%) for a slight increase (0.25%) in the bandwidth cost of the CDN.

Our contributions in this work include a unified framework to jointly optimize mapping, placement, and disk allocation while including the impact of cache misses. We offer workload characterization using real datasets. This work starts with a relatively simple model of performance based on the distance or latency between the client and edge server, and edge server and origin server. But, in reality, end-to-end performance depends on a variety of other factors, including performance problems at the edge server, congestion in the network, or problems at the client machine. To uncover this wide range of problems along the end-to-end path, we present our next tool, Diva, to diagnose roots of poor performance, focusing on one content provider—in this case, a video provider.

DIVA[48]: Diagnosis of Internet Video Anomalies. Despite the growing popularity of video streaming over the Internet, problems such as re-buffering (i.e., when the video stream stalls) and high startup latency (i.e., when the user clicks to play the video but faces a delay) continue to plague users. Diva uses a chunk-based instrumentation methodology to model the end-to-end path of video delivery system and diagnose performance problems within this path, for every chunk, and across the chunks in a video session. We instrumented both the CDN servers and the client video player, while also collecting frequent snapshots of TCP variables from the server network stack, allowing us to construct an end-to-end view of video performance for the first time. With this unique visibility into the causes of performance degradation, Diva uncovered a wide range of performance problems that were not known before, including an asynchronous disk-read timer, impact of cache misses at the server, high latency and latency variability in the network, buffering delays and dropped frames

at the client. Looking across chunks in the same session, or destined to the same IP prefix, we see how some performance problems are relatively persistent, depending on the video's popularity, the distance between the client and server, and the client's operating system, browser, and Flash runtime. Our findings were used in Yahoo to optimize the video performance.

Our contributions in Diva include (1) a diagnosis tool that for the first time joins the dataset from the server side with client and network side, and (2) deploying it in an operational setting in the wild (Yahoo player and CDN). Diva can accurately pinpoint the sources of problems in the video delivery path. Still, Diva faces a limitation in the frequency of network snapshots due to high overhead and storage limitation. Collecting fine-grained network performance data on the server is costly. To work around this limitation, we propose our next tool, Dapper.

Dapper[47]: Data-plane Performance Diagnosis of TCP A TCP connection may have bottlenecks at the sender, receiver, or the network. In Table 1, we present several examples of performance problems that may arise at each location. With many applications and TCP congestion control variants in use today (e.g., Cubic [51], Taho and Reno [56], New-Reno [54], Vegas [16]) it is challenging to decide what minimal set of metrics to collect that are both affordable (i.e., does not consume a lot of resources) and meaningful (i.e., helps in diagnosis). It is also essential to diagnose such TCP performance problems in a timely manner. Offline processing of logs is slow and inefficient, and collecting TCP metrics at end-hosts requires patching kernels (e.g., to use tools such as Web10G [105]). It is also costly to frequently snapshot TCP metrics from the kernel (e.g., tcp_info), as we mentioned in the limitations of Diva above. In addition, relying on network information offered by the kernel limits the measurement flexibility (e.g., kernel collects smoothed averages of key statistics, rather than individual samples, as we discuss it in more detail in Chapter 3).

Instead, our tool, Dapper, analyzes TCP performance in real time near the edge servers at line-rate (e.g., at the NIC or the edge switch). Dapper determines whether a TCP connection is limited by the sender (e.g., a slow application that is bottlenecked elsewhere such as disk), the network (e.g., congestion), or the receiver (e.g., small receive buffer). We use P4 [14] which is a language for programming the data plane of network devices to prototype Dapper and evaluate our design on real and synthetic traffic. To reduce the data-plane state requirements, we perform lightweight detection for all connections, followed by heavier-weight diagnosis just for the troubled connections.

Our contributions in this work include a data-plane based system for diagnosing TCP connections' performance problems, at line-rate, that can detect the source of problems. Our measurement in Dapper is "inference-based", meaning we do not require cooperation from the end-hosts at either side; Dapper makes a diagnosis simply by analyzing the packet stream at the edge of the network.

In the next section, we outline how the three pieces of the thesis work together in a control loop.

1.3 The Steps in Managing the Performance

To efficiently and uniformly manage the performance and diagnosis problems across the CDN platform, we present a control loop that consists of three steps. Figure 1.3 shows the control loop and how these three steps fit in together. Let us explain these steps, in order:

1. Measurement: Measurement is the first step in detecting, diagnosing, and managing the CDN performance. Measurement could mean different things, for example, in CAM we collect server request logs from edge servers with Akamai. In Diva, we introduce new instrumentation infrastructure and performance metrics across the end-

to-end path of the video delivery system and join the server and client-side datasets to understand what impacts the performance of the delivery path. In Dapper, we are interested in collecting finer-grained TCP statistics to find out if the TCP performance is limited by the sender, receiver, or the network. Thus, we build a new inference-based measurement of TCP from the edge device.

2. Analysis: The next step after measurement is Analysis. Our datasets are at industry scale. We instrumented over 85 CDN servers in Diva and analyzed more than 500 million video chunks. In CAM, we study hundreds of servers with billions of request logs. These huge datasets are analyzed on Hadoop and Spark clusters, via MapReduce jobs.

3. Action: The final step is taking appropriate actions; which can differ based on the goal of each project. These actions could include (1) optimizing the CDN configurations (CAM), (2) using the results of analysis to come up with a better system design (Diva), or even (3) triggering finer grained measurements to pinpoint the source of poor performance (Dapper).

The remainder of this thesis is organized as follows: Chapter 2 presents CAM, the unified framework for resource allocation within a CDN. Chapter 3 describes Diva, the end-to-end chunk-based performance diagnosis system that finds the roots of performance problems causing poor user QoE. Chapter 4 presents Dapper, the real-time TCP performance diagnosis based on inference-based measurements that runs at line-rate at the edge. Chapter 5 discusses how the three systems work together and concludes the thesis.

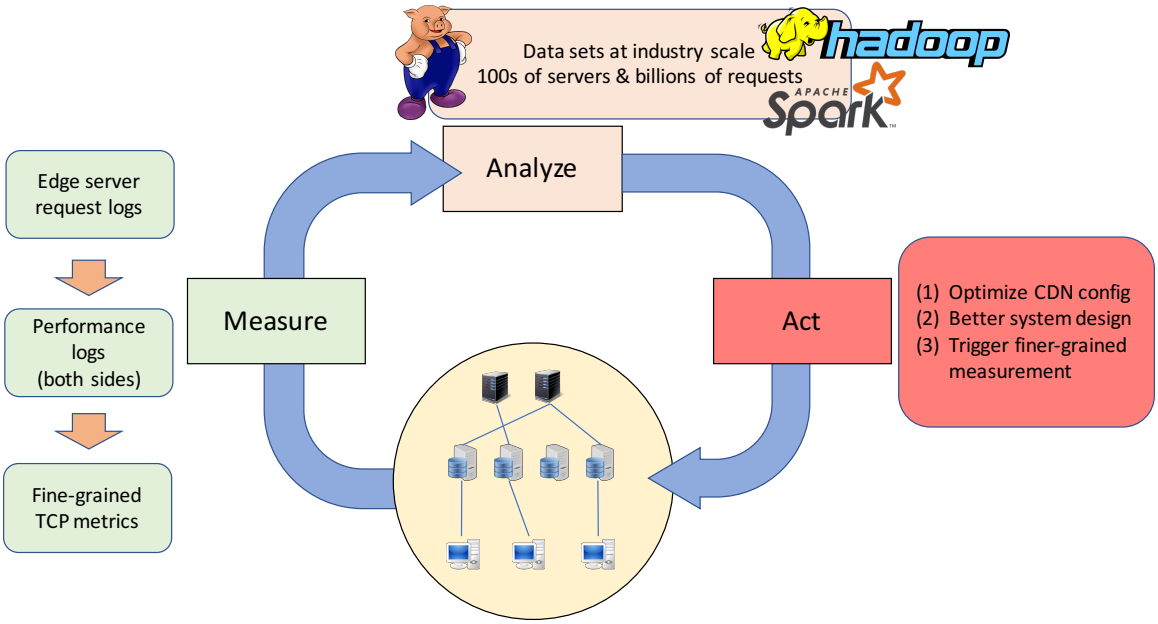


Figure 1.3: Our control loop consists of three steps: (1) Measurement of the system, (2) Analyzing the datasets, usually via big data analysis frameworks, and (3) Acting based on the results.

Chapter 2

Cache-Aware Mapping

Most of the web traffic is served through CDNs; Akamai alone was responsible for serving up to 30% of traffic in 2016 [29]. Recent work has shown the negative impact of cache misses on the quality of experience of end-users [48, 91]. In addition, cache misses are costly for CDN operators since they increase the consumed bandwidth; cache misses also make content providers (CP) unhappy by increasing the load on origin servers. Nevertheless, the impact of cache misses has not been explicitly modeled in a unified way in CDN's resource allocation algorithms (e.g., mapping, placement, and disk partitioning).

In this chapter, we propose the Cache-Aware Mapping (CAM), a tool to allocate resources across the distributed edge servers in a CDN to jointly minimize network latency and cache miss rates and strike a balance between competing goals of a CDN (i.e., performance and cost). CAM offers a unified model that optimizes performance while explicitly modeling the impact of cache misses and without making simplifying assumptions about the traffic (e.g., assuming the traffic always follows a Zipfian distribution). We offer a unified model for CDNs to solve three essential problems of content placement, mapping, and disk allocation. Since the joint optimization is a non-linear non-convex problem, we pick one out of the three problems to focus on:

the disk allocation problem. We chose to optimize the disk partitioning problem for a few reasons: (1) Akamai already has a mapping and placement algorithm, but the disk optimization is done using an approximation to the LRU policy [67]. This optimization aims to minimize the total cache miss rate, without taking into account that some cache misses are costlier than others. (2) We are interested to know if we can improve the performance by only managing the disk efficiently without changing the current mapping and placement systems, and (3) disk management is a decentralized process that is run on each edge server, hence it can be deployed incrementally, but changes to mapping and placement are global and affect the entire platform at once. Using server request logs from a commercial CDN (Akamai), we show that even without changing the current mapping or placement algorithms, by only managing the disk efficiently, we can enhance the performance—measured as the average distance or network latency from clients to edge servers, and from edge servers to origin servers—significantly (17%) for a slight increase (0.25%) in the operational cost—measured by the bandwidth consumption.

2.1 Introduction

Let us start with an overview of how a CDN maps a content request to an edge server. In this setting, a client is making a request for a specific content from a content provider (CP), and the CP is using the services of a CDN. When the client makes a DNS request for the CP, instead of getting the IP of the CP it gets the IP address of one of the CDN's edge servers. This mapping or assignment of clients to edge servers is usually done via a global load balancer (GLB) [1]

[1]There are CDNs such as CloudFlare [31] that use anycast [10] for mapping. A DNS-based mapping offers fine-grained and near-real time control over the mapping, but requires considerable investment in infrastructure [21], whereas an anycast mapping is often cheaper but offers minimal control. In this work, we use the DNS-based model to be consistent with the Akamai ecosystem.

When GLB maps a client to an edge server, it tries to achieve the *best performance at a reasonable cost*, which means that (1) the edge server should be able to handle the request (i.e., the load and liveness of server is considered), (2) the edge server should be nearby (i.e., proximity or network performance is considered), and (3) the edge server should avoid going to to the origin server as much as possible (i.e., caching is considered).

The majority of the related work in replica selection methods have focused on balancing the proximity and the load (goals 1 and 2 above); however, less attention has been given to maximizing cache hits. More importantly, the impact of cache misses have often been ignored in these solutions [107, 85, 29, 89, 7, 50, 22, 82]. In addition, the previous work on cache replacement algorithms [9, 79] make simplifying assumptions such as assuming that the workload belongs to a single CP, or the content popularity follows a specific distribution, in particular Zipfian [17, 81]. Instead, in CAM we want to explicitly model the impact of cache hit rates on performance and cost, and show how we can balance their trade-off by optimzing the allocated disk per CP.

2.2 Impact of Cache Misses

Cache misses have a negative impact on all involved parties, as shown in Figure 1.2. First, cache misses are costly for the CDN. If content does not reside in the local disk of the edge servers and needs to be fetched from the origin servers of a particular CP, the CDN has to pay the ISP(s) for the consumed bandwidth. A higher cache miss rate increases the usage and escalates the need for a higher-capacity connection. Second, cache misses at the edge make CPs unhappy, because it means more load is sent to the origin servers of the CP [2], which sometimes cannot handle this high load. Third, cache

[2]For simplicity, we are ignoring the hierarchy within the CDN but in reality CDNs have multiple layers of cache servers

misses impair the end-user performance. Recent work [48] shows that cache misses increase the average latency of serving the content by 10X. We conducted a 10-day study in Akamai in cooperation with a large video provider, where we analyzed and correlated the client-side user QoE data with server-side request logs. We observed that *requests that are associated with re-buffering at the client-side have a higher cache miss rate at the server-side.*

This high impact of cache misses on all involved parties is the motivation behind CAM: To enhance performance and cost *while including the impact of cache misses*, based on real workloads. In the next section, we overview the problems a CDN faces, and the decisions a CDN makes. In future chapters, we will incorporate the impact of cache misses in all three decisions.

2.3 CDN Resource Allocation Problems

Before discussing the model, we need to understand the resource allocation problems a CDN faces. There are three general resource allocation decisions that a CDN makes, where we believe the impact of cache misses needs to be explicitly modeled:

1. Placement: Content placement is the decision of "which edge servers should be able to serve the content, for each content provider". Content placement decisions are made on a long time-scale and are static. The solution to the content placement question for each CP is a set of binary values for each edge server, where 1 means that CP will be served from that edge server, and 0 means that CP will not be served from that edge server. Content placement is solved at the granularity of a CP, which may seem coarse-grained in contrast to placement of individual objects, but this is done to (1) make the platform scale better, and (2) each CP is associated with a domain that DNS resolves. A group of content (or objects) —the CP's library—would have the same domain name. Note that content placement is essential for CDNs since it's

not always ideal to serve a CP from all the edge servers, because spreading the load among more servers lowers the cache hit rates; in addition, after some point there is no performance gain in adding more edge servers. We will elaborate more on this in the next section with an example.

2. Mapping: Mapping is the decision made by the GLB to see "for each client request of a content provider, which one of the replicas should serve it?". Mapping takes the placement constraints into account (i.e., won't send the request to a server not included in the content placement set as described above), and uses real-time load and latency data to ensure the chosen edge server is (1) alive, (2) can handle the extra load, and (3) is proximal, i.e., has a good network performance to the client.

The solution to the mapping problem can either be one chosen server, or a set of servers (split-load). In this work, we assume the GLB can handle split-load mapping, meaning, not all requests from a certain client cluster need to go to a single server, rather, the load can be divided among several servers. This assumption is reasonable for our system because GLB operates at client prefix level, so it is possible to direct individual clients within a prefix to different edge servers. In addition, the DNS responses contain Time To Live (TTL), meaning after its expiration a new request to the GLB is made, therefore it is possible for the GLB to split the load across multiple servers on a longer time-span than a TTL. Hence, the numerical solution will be a set of values in $[0, 1]$ range, where their sum of values across a client cluster equals 1 (i.e., all requests of that cluster are satisfied).

2. Disk Allocation: Once the CDN has decided which servers should serve each CP (i.e., content placement), and which requests to send to each server (i.e., mapping), each edge server ends up serving a group of CPs. The CDN needs to decide how to manage or allocate the disk among these CPs on each edge server (i.e., the disk allocation problem).

To understand how each of these three decisions interacts with the cache performance, and thus the CDN goals, let us walk through two simple examples. The first example will focus on content placement and mapping across the servers, and the second one will focus on disk allocation within a server.

Example 1: Placement and Mapping

Consider a simple setting that a CDN has three servers across the US (Figure 2.1), with three CPs and clients located in three clusters. If the goal of the CDN is to *minimize network latency* (or the distance) between the clients and the edge servers, then it should serve every CP at every edge server, i.e., the clients always get mapped to the closest edge server (Figure 2.2a). In this solution, the content placement decision includes all servers. However, there is a major problem with serving every CP from every edge server: it can lower the cache hit rate. When an edge server serves more CPs, the competition for the shared disk (cache) increases; resulting in the objects of different CPs evicting the contents of other CPs from the shared cache, and hurting the cache hit rate.

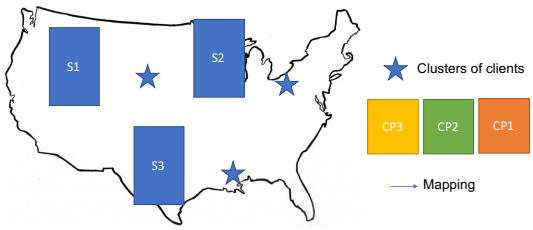
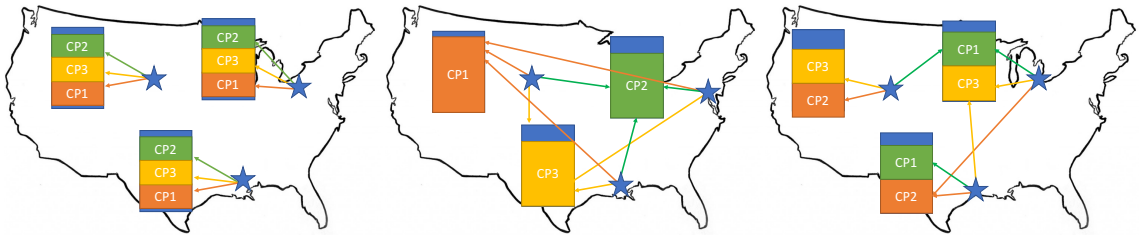


Figure 2.1: A placement and mapping example.



(a) Every CP on every edge (b) Dedicated edge per CP (c) A balanced solution

Figure 2.2: Three different solutions of the mapping and placement example.

If instead of minimizing the clients latency to servers, the CDN chooses to *maximize cache hit rates*, we'll reach a different solution shown in Figure 2.2b. In this solution, each CP has a dedicated edge server which minimizes the competition as the cache is not shared; thus, achieves maximum per-CP cache hit rate. However, in this solution clients are mapped to dedicated servers further away, which may hurt the client-perceived performance.

These two solutions show the two extreme objectives of the optimization. The first tries to optimize for performance (latency) and the second tries to optimize for cost (cache hit rate). However, our goal in this work is to show that we can capture both performance and cost in a joint optimization, and strike a balance between them. Figure 2.2c shows a middle-ground solution where the cache hit rates are not as bad as the first solution since there is less competition between CPs at each server, but the performance is not as bad as the second solution either because the clients are not mapped that far. The observation we make from this example is that there are *diminishing returns* in both aspects of this optimization: serving each CP from a dedicated edge server enhances the cache hit rate because the disk can fit the tail of its library. Still the tail of the distribution is unpopular, so the performance gain becomes negligible eventually. Similarly, serving every CP everywhere offers the most proximal server to clients, but the network performance may already be adequate with fewer locations.

We can see theses diminishing returns from real datasets: (a) Figure 2.3 shows the performance gain for a particular CP as we increase the number of edge servers from 10 to 100 unique locations. The y-axis is the reduction in sum of traveled distances if the CP has x edge servers, normalized by the sum of distances if there were only 10 edge servers. We can see that initially there is a performance gain as we increase the number of servers, but by increasing number of edge servers past 40 servers the gain becomes negligible. This is because this particular CP's clients are clustered around

about 50 major locations within the US. (b) Figure 2.4 shows the cache miss rates of a particular CP as we increase its disk capacity. The x-axis shows how many units of the CP's library we can fit in the cache. Initially by allocating more disk to this library, cache miss rate is significantly reduced. However, once we get to the tail of the distribution (i.e., less popular items) the gain in cache miss rate by adding more disk becomes negligible.

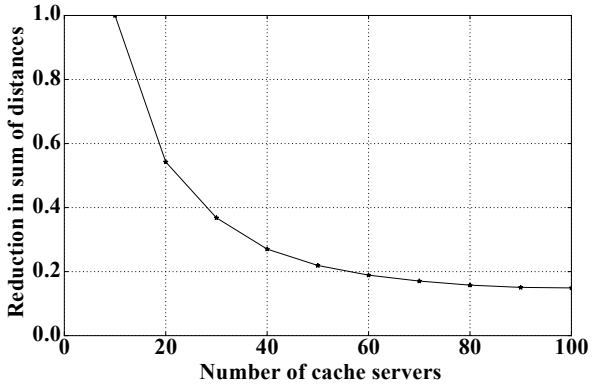


Figure 2.3: Diminishing gain in performance (distance) as number of caching servers increases.

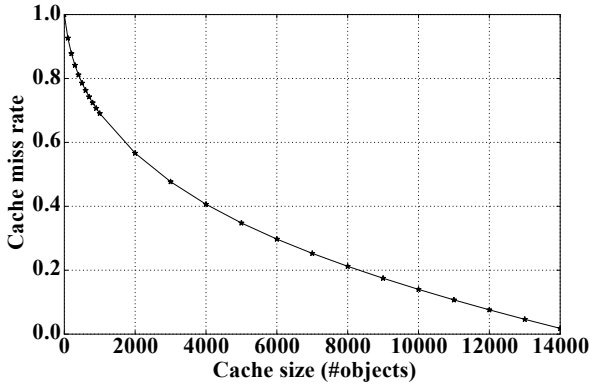


Figure 2.4: Diminishing gain in cost (cache misses) as size of disk (MB) increases.

Example 2: Disk Allocation

The previous example showed us how mapping and content placement affect the CDNs' goals, now lets zoom into one of these edge servers and see how the disk should be managed: If the goal of the CDN is "minimizing the overall cache miss rate", then

a shared cache that evicts the least frequently used (LFU) item or the least recently used item (LRU) naturally keeps items in the cache that enhance the cache hit rate.

However, our goal in this work is "minimizing the *impact* of cache misses", not cache miss rates themselves. Figure 2.5 shows the *impact* of cache misses for clients of two different CPs, served from the same edge server. We can see that cache misses for CP 1 are more harmful than CP 2 because the origin server of CP 1 is further away. This means the clients of CP 2 must wait longer on a cache miss to fetch the content from a further away origin server. The observation that we make from this example is that the impact of cache misses is different among CPs, hence "the ability to partition the cache is essential". By *partitioning* the cache, we mean assigning an amount of disk size that is reserved for each CP, based on the popularity distribution of the CP as well as the latency to its origin. Related work such as [30] make similar observations with formal proof that partitioning the cache yields better performance compared to sharing it.

While for a CDN serving multiple CPs partitioning the cache is better than sharing it, we still must decide the granularity of partitioning. One can imagine per-object partitioning where a"performance toll" is associated with each object that reflects the distance (or latency) to the origin server upon cache misses. However, a per-object partitioning makes it challenging to model and enforce fairness among CPs. Instead, we believe it is best for a CDN to partition the cache per CP, to manage the cost and performance while being able to enforce fairness between different CPs. In addition, per-CP partitioning makes the toll estimation at the edge servers simpler and avoids redundancy since the objects of the same CP are fetched from the same origin and incur a similar toll on user-perceived performance.

To summarize, in this section, we explained the three essential decisions a CDN makes: placement, mapping, and disk allocation. We demonstrated with two examples how we can strike a balance among the competing goals of a CDN (performance

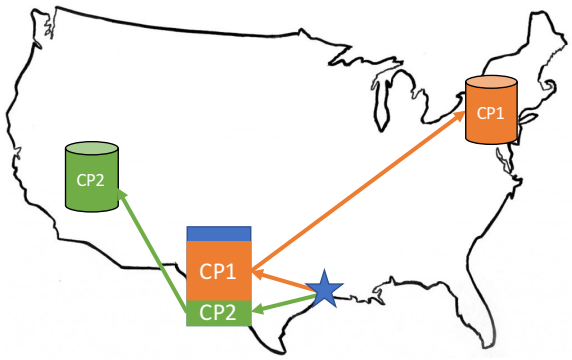


Figure 2.5: A disk allocation example: Origin servers of CP 1 are further away from the edge server than CP 2, making its cache misses more costly.

and cost). We observed that both of these objectives, if considered alone, have diminishing returns. We also showed that the impact of cache misses on clients (i.e., the extra latency) differs among different CPs, therefore, the ability to partition the cache is essential in managing the impact on end-users. In the next section, we explain how to capture these three problems in a unified framework, while modeling the impact of cache miss rates explicitly in the model.

2.4 Unified Performance Model

In this section, we will first explain the metrics that matter to a CDN(i.e., goals). We then show how to capture all the three problems that a CDN faces into a single framework, and how to optimize this model according to these goals (e.g., best performance). For simplicity we use geographical distance, but the cost matrices can be replaced with network latency, throughput, or even custom metrics defined by a CDN (e.g., throughput over latency). Let us begin by explaining the metrics that CAM focuses on:

2.4.1 Goals of a CDN

There are three major metrics that a CDN cares about, and our goal is to model them with CAM; These metrics are:

1) Performance: CDNs care about user-perceived performance, e.g., the perceived latency or throughput. The performance goal of CAM is to minimize sum (or average) of distances (or latencies) for clients to servers. Note that when there is a cache miss, there is an additional distance (or latency) from the edge caching server to origin to be considered.

2) Cost: Cost is a measurement of how much a CDN pays for operations. to pay ISPs for consuming bandwidth to serve clients. This is less of an issue for serving a client's request since it brings profit, however, cache misses require spending money to fetch the content first from origin servers and then serving it to the client, hence cache misses increase the net operational cost. Lowering cache misses helps reduce the cost of running the CDN platform.

3) Fairness: CDNs strive to preserve a notion of fairness among CPs. Fairness is complicated to model. For example, a CP may naturally have more cacheable content, or have a smaller library size that fits well in any cache size, or have more popular content. Therefore, instead of optimizing directly for fairness, we (1) show the impact of optimized performance solution on fairness, and (2) add fairness constraints to the framework to ensure the solution we reach at does not treat different CPs unfairly. There are different criterion to measure the fairness, here we use the relative amount of resources (e.g., the disk) allocated to each CP as a notion of fairness. We use Jain's fairness formula (shown below) to measure the fairness, where x_i is the fairness criteria, which is the relative amount of resources dedicated to CP_i:

$$J(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} {x_i}^2} \tag{2.1}$$

Next, we will outline the inputs to the system, along with the notation used in the unified framework.

2.4.2 Inputs and Notations

Figure 2.6 summarizes the notation used in this section. A content provider (CP) origin server is marked with"i", CDN edge server with "j", and client cluster with "k" [3].

The distance (or latency) between the origin server of CP_i and the CDN's edge $server_j$ is shown with $b_{i,j}$. Similarly, $a_{j,k}$ captures the distance between edge server "j" and client cluster "k". In addition, the $c_{i,k}$ value shows the demand for CP_i from $client_k$. In this model, we use variable $x_{i,j,k}$ to represent the portion of the demand of $client_k$ for CP_i, that is served by edge $server_j$. Finally, the $m_{i,j}$ variables show the cache miss fraction of CP_i on edge $server_j$. Table 2.1 shows the list of notations used in the model.

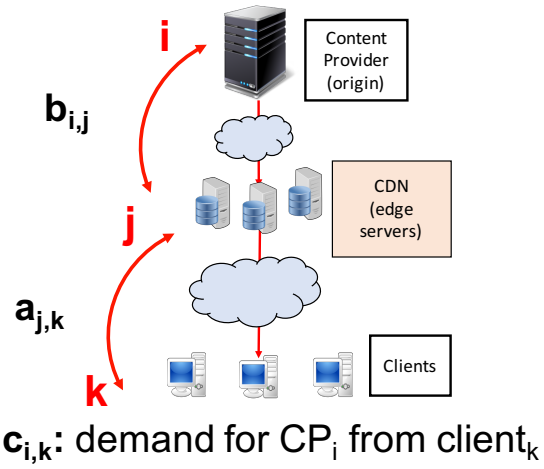


Figure 2.6: Notations used in the unified performance model

Using these notations, we are now equipped to discuss the unified performance model.

[3]To reduce the complexity of the problem and make the optimization scale better we have clustered clients based on their geographical location and AS

Table 2.1: Notations used in the unified performance model

Notation	Meaning
i, j, k	A CP, a CDN edge server, and a client cluster respectively.
$a_{j,k}$	The path cost (e.g., distance or latency) of accessing edge server j from client k.
$b_{i,j}$	The cost of path from edge cache j to origin servers of CP i.
$c_{i,k}$	The demand for CP library i from client k.
$x_{i,j,k}$	The fraction of client k's demand for CP library i mapped to edge server j.
$m_{i,j}$	The miss rate of CP library i at edge cache j.
$d_{i,j}$	Disk space allocated for CP library i at edge server j.
k_j	The capacity of edge server j, in terms of how much demand it can take.
$disk_j$	The disk capacity of edge server j.
$l_{i,j}$	The demand for CP i served by edge server j.
U	set of clients

2.4.3 Unified Performance Model

Using the notation as explained above, the performance of the requests from $client_k$ for CP_i served from $edge_k$ can now be captured via the following equation:

$$perf = x_{i,j,k} \cdot c_{i,k} \cdot (a_{j,k} + m_{i,j} \cdot b_{i,j}) \tag{2.2}$$

This equation captures two terms: First, for the portion of the demand ($x_{i,j,k} \cdot c_{i,k}$) from the client to the edge server for a particular CP, all of such requests travel the distance to the edge server ($a_{j,k}$). In addition to that, the portion of this traffic that experience a cache miss at the edge server ($m_{i,j}$) will incur an additional distance (or latency) of fetching the content from the CP's origin server ($b_{i,j}$).

By summing Equation 2.2 over all client clusters, all edge servers (shown as S), and all CPs, we will reach to Equation 2.3. This equation represents the performance of the entire system, measured as total traveled distance (or total latency experienced by end-users). Dividing this equation by the total demand gives us the

average traveled distance (or average latency), considering the extra performance toll of communication with origin servers due to cache misses.

$$perf = \sum_{i \in CP} \sum_{j \in S} \sum_{k \in U} x_{i,j,k} \cdot c_{i,k} \cdot a_{j,k} + \sum_{i \in CP} \sum_{j \in S} \sum_{k \in U} x_{i,j,k} \cdot c_{i,k} \cdot m_{i,j} \cdot b_{i,j} \tag{2.3}$$

The constraints of this model include:

- The assigned demand to each edge server cannot exceed the capacity of that server, denoted by k_j:

$$\textit{for all } j\textit{:} \quad \sum_{i \in CP} \sum_{k \in U} x_{i,j,k} \cdot c_{i,k} \leq k_j \tag{2.4}$$

- Demands can be partially assigned:

$$\textit{for all } i, j, k\textit{:} \quad 0 \leq x_{i,j,k} \leq 1 \tag{2.5}$$

- Demands of each client cluster for each CP should be satisfied, regardless of which servers take that responsibility:

$$\textit{for all } i, j\textit{:} \quad \sum_{k \in S} x_{i,j,k} = 1 \tag{2.6}$$

- The total disk allocations given to CPs on an edge server cannot exceed the disk capacity of that server (we will discuss how to allocate $d_{i,j}$ values in the next section):

$$\textit{for all } j\textit{:} \quad \sum_{i \in CP} d_{i,j} \leq Disk_j \tag{2.7}$$

Our unified performance model captures all three problems that CDNs face, while incorporating the impact of cache misses in them. Let's see how this framework captures the three problems in one model:

1. **Placement**: The solution to the placement decision is captured in $x_{i,j,k}$, if x is always zero for clients of a CP on an edge server (i.e., never mapped there), the library of the CP will not be placed at that edge server, while a nonzero x indicates the library of the CP can be placed at this edge server, and requests can be mapped there.

2. **Mapping**: the solution to the mapping decides how much of the demand from each client cluster should be directed to each server. This is captured directly in $x_{i,j,k}$ values. Note that $x_{i,j,k}$ is a variable in the range $[0, 1]$ because we assume the CDN can split the load.

3. **Disk Allocation**: Finally, the solution to the disk allocation problem is (indirectly) captured in the miss rate of each CP on each edge server: $m_{i,j}$. $m_{i,j}$ is a complex function of various factors, including the size of the library of CP_i, the popularity distribution of objects in the library of CP_i, the size of the disk on $server_j$, and if $server_j$ is hosting other CPs (i.e., cross traffic) with their respective library sizes and popularity distributions. In the next section, we will explain how we model the miss rate of each CP as a function of the allocated disk.

In addition to capturing these three CDN problems in the performance model, our unified model has the flexibility of including *fairness* and *cost* as constraints in the model:

Cost: The associated bandwidth cost of CDN is a function of miss rate, as more cache misses cause more bits to be pulled into the edge servers. We model the cost

constraint by the total number of cache misses across the edge servers for all CPs ($l_{i,j} = \sum_k x_{i,j,k} \cdot c_{i,k}$ is the amount of demand for CP_i served by edge $server_j$):

$$Cost : f(total\ cache\ misses) = \sum_{i \in CP} \sum_{j \in S} m_{i,j} \cdot l_{i,j} \tag{2.8}$$

Fairness: As explained before, the fairness of a solution can be evaluated by comparing the cache miss rates of different CPs, before and after partitioning the cache. To enforce a fair solution, we can bound the cache miss rates (i.e., acceptable margin of cache miss rates). This will direct the optimization to a fair solution where we avoid situations like dedicating the entire cache to one CP with the furthest away origin servers. We can accomplish this via enforcing a lower bound on the size of the allocated disk ($disk_{min}$) to each CP, which ensures a maximum cache miss rate.

$$disk_{min} \leq d_{i,j} \leq disk_j \tag{2.9}$$

To find the exact amount of $disk_{min}$, we need to understand the relationship between the disk size and the CP's library, for every CP, which is the topic of the next section (Section 2.5):

2.4.4 Solving the Unified Performance Model

To solve the joint optimization, we begin by assuming the distance (or latency) costs are constant between the client locations and edge servers or between edge servers and origins (i.e., $b_{i,j}$ and $a_{j,k}$). While this assumption is true for distance, network latency may change throughout the day due to routing changes, network congestion, or the server delay itself. We will discuss how our framework can handle such changes in Section 2.8.

With this assumption, the first part of Equation 2.3 becomes convex and linear with respect to x; The second part of the equation is however non-linear and non-

convex, because it contains the product of two variables, x and m ($x_{i,j,k} \cdot c_{i,k} \cdot m_{i,j} \cdot b_{i,j}$). Via numerical analysis, we show that $m_{i,j}$ is a non-linear but convex function of $D_{i,j}$ (see next section); however, since the product of two convex functions is not necessarily convex, we cannot use convex optimization tools. The joint optimization can still be solved via search optimization methods (e.g., gradient descent, simulated annealing) where the solution converges to a local minimum, but it cannot be guaranteed that the global minimum will be reached. Because of this limitation, we decided to focus on only one of the three main problems. Since most CDNs currently do not partition the disk or do not take into account that some cache misses are more costly than others [67], we focus on the disk allocation problem. Our goal is to answer *if we can improve the performance by "only" managing the disk, with the current mapping and placement algorithms in place.*

Focusing on the disk allocation problem alone makes the optimization problem simpler than the joint optimization. Because the mapping and placement is given, we sum over all the clients' demands ($c'_{i,j,k} = x_{i,j,k} \cdot c_{i,k}$) to get the portion of demand of each CP served by each edge server ($l_{i,j} = \sum_k c'_{i,j,k}$). This effectively reduces the dimensions of the problem from three (client clusters, edge servers, and CPs) to two (edge servers and CPs), as shown below in Equation 2.10:

$$perf = \sum_{i \in CP} \sum_{j \in S} \sum_{k \in clients} c'_{i,j,k} a_{j,k} + \sum_{i \in CP} \sum_{j \in S} l_{i,j} \cdot m_{i,j} \cdot b_{i,j} \tag{2.10}$$

In this simplified version of the unified model, the optimization variables are $d_{i,j}$, or the amount of disk allocated CP_i on edge $server_j$. In addition, under a given mapping the first term of the optimization—the mapping distance between clients and edge servers—becomes constant; hence, the optimization only minimizes the second term—sum of the distances (or latency) from edge servers to origin servers for cache misses.

$$perf = \sum_{i \in CP} \sum_{j \in S} l_{i,j} \cdot m_{i,j} \cdot b_{i,j} \tag{2.11}$$

This new optimization formulation, as shown in Equation 2.11, is more tractable and is convex with respect to $d_{i,j}$ under one condition that $m_{i,j}$ is convex with respect to $d_{i,j}$, which would guarantee that a convex optimization solver will find the global minimum. Over the next section, we model the cache miss rates and understand the relationship between $m_{i,j}$ and $d_{i,j}$ and prove its convexity.

2.5 Modeling Cache Miss Rates

The goal of this section is to model the cache miss rate as a function of available disk, or $d_{i,j}$. The cache miss rate depends on a few factors. First, the *cache eviction policy* that is in use on the edge server. For example, whether the cache evicts the least recently used item (i.e., LRU) or the least frequently used item (i.e., LFU). Most CDNs, including Akamai, use the LRU cache eviction policy on edge servers [29]. Second, the amount of *disk size* available for caching. Naturally, the larger the disk is, the more items can stay in the cache, which increases the time an object stays in the cache until it is evicted; this time is referred to as the "eviction age" of the object. A larger disk allows the tail of the distribution (i.e., the less popular objects of the library) to stay in the cache. Third, the *library size* of a CP—how many objects—and the *size of objects* in the library play a role in the cache miss rate as well. For example, a small library fits in a small cache size on the edge servers, and a library with smaller objects can fit more items in a fixed size cache. Finally, the *popularity distribution* of the library impacts the caching efficiency [43]. For example, consider two libraries A and B with similar sized objects each containing the same number of objects. Library A has a "popular-heavy" distribution where the top 10% of items receive about 90% of requests. Library B has a uniform distribution where

all items have a similar request rate. If each library gets a limited quota of the disk, for example only enough to fit 10% of their items, it is clear that the caching efficiency will be better for library A and its cache miss rate will be lower because its objects are evicted from the cache less frequently. Let us present a formal definition of "popularity distribution" of a CP before moving on to the cache miss rate model.

Popularity Distribution: Cache performance depends crucially on the popularity distribution of a library, shown with $q()$. It is usual to order objects in order of deceasing popularity such that $q(1) \geq q(2) \geq ... \geq q(N)$.

Zipfian Law: With the convention as explained above, the most frequently observed popularity law [17, 25] is the generalized Zipfian law of: $q(n) = 1/n^{\alpha}$, with $\alpha > 0$

Based on the discussion above, what we need from a cache model is a "predictive model for the cache miss rates, as a function of disk space available, per library". It is important to know that we *cannot analytically derive this model*, unlike some of the previous work [81, 71]; because based on a large scale characterization study with Akamai we found that *a lot of workloads do not follow the Zipfian law*; In fact, we observed a variety of popularity distributions across different CPs, a few examples are shown in Figure 2.7.

Each figure shows the popularity distribution of objects within an example CP. The x-axis shows the ranks of objects; objects are sorted from the most popular ($rank = 0$) to the least popular; the y-axis shows the relative popularity of that item, which is fraction of requests of the library that belong to that item. The length of the x-axis is different because the CPs have different library sizes. These figures show us that the simplistic assumption that all workloads are Zipfian does not adhere to the ground truth, as obtained from the characterized dataset, and should be avoided in our model.

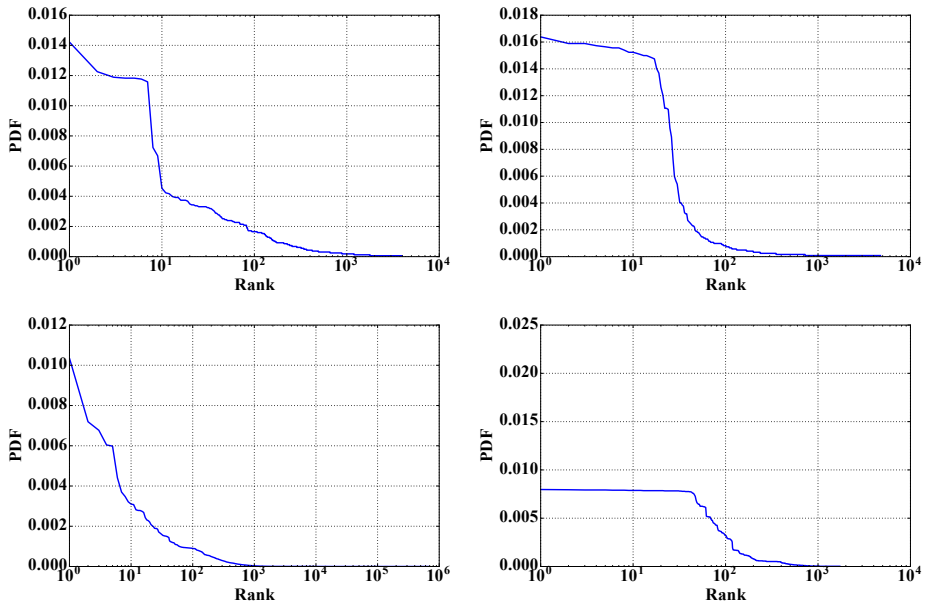


Figure 2.7: Popularity distribution of a few CPs, objects are sorted from most popular (rank=0) to least popular. We can see that the common assumption of zipf-like popularity distribution does not hold here.

Unfortunately, evaluating the performance of the cache under different workloads is hard, considering that the computational cost to exactly analyze just a single LRU cache grows exponentially with both the cache size and the number of contents. To resolve this issue, several approximations have been proposed over the years [35, 27, 45, 43, 84] which can accurately predict cache performance at an affordable computational cost. In the next part of this section, we discuss the existing models and evaluate the accuracy vs. overhead trade-off for each.

2.5.1 Approximation Models for Cache Miss Rate

There are various ways to model an LRU cache; To understand these models we need to equip ourselves with an understanding of the Independent Reference Model (IRM), which is the standard approach adopted in the literature to characterize the pattern of object requests arriving at a cache [45, 33].

Independent Reference Model: The IRM, as first introduced by [33] is based on the following fundamental assumptions: i) users request items from a fixed library of N objects; ii) the probability $q(n)$ that a request is for object n, $1 \leq n \leq N$, is constant (i.e., the object popularity does not vary over time) and independent of all past requests. Under this model the references to stored objects are independent random variables, generating an independent and identically distributed (i.i.d.) sequence of requests.

Our goal is to choose a model that offers a sweet spot between *accuracy* and *computational overhead*. Our cache sizes are in the order of Tera-bytes, hence a computationally expensive model does not fit our purpose well. Still, we want the model to provide adequate accuracy in results. Here, we briefly present a few of caching models to see which one fits our needs the best. We invite the readers to see the related work at each part for a more detailed explanation. We will indicate the size of available cache (or disk) with C:

- **LFU:** The least frequently used cache eviction policy stores the C most popular items of the library in the cache. LFU is known to provide optimal performance under IRM. Under the conventions described above, the cache miss rate under this model can be described with the following Equation:

$$m(C) = 1 - \sum_{0 \leq i \leq C} q(i) \tag{2.12}$$

  This model needs to the popularity distributions of each object ($q(n)$) to estimate the cache miss rate, given the cache size. While this LFU model is simple and has low computational overhead, it is not accurate enough to describe the LRU cache used in the CDN edge servers.

- **Che's approximation:** Che's approximation of modeling the LRU cache is a very well known method [43, 45], as first proposed by [27].

The probability a request is for object n, for $1 \leq n \leq N$, is proportional to some popularity $q(n)$, independently of all past requests. The hit rate $h(n)$ for object n, i.e., the probability this object is present in the cache, is approximated by:

$$h(n) \approx 1 - e^{-q(n)T_C} \tag{2.13}$$

Where T_C is called "the characteristic time", and is the unique root of the following equation:

$$\sum_{0 \leq n \leq N} (1 - e^{-q(n)t}) = C \tag{2.14}$$

The characteristic time or T_C can be approximated as the time at which exactly C unique objects have been requested. With this model the cache miss rate can be modeled as:

$$m(C) = \sum_{0 \leq n \leq C} e^{-q(n)T_C} \tag{2.15}$$

Che's approximation has been shown to be a very accurate model of the cache through extensive experiments [43].

- **Stack Distance:** LRU stack distance, as first defined by [69], is the number of distinct data accesses between two consecutive accesses to the same location in cache. Consider the following access pattern to a cache:

  a, b, a, b, b, c, d, a, b, c

  The number of unique objects between two consecutive requests to each item is shown in the Table 2.2 below. Notice when an object is requested for the first time it is fetched from the origin (i.e., cache miss), shown with a distance of ∞.

Table 2.2: Stack distance of the items in the sequence pattern

Item	**Stack Distance**
a	∞, 1, 3
b	∞, 1, 0, 3
c	∞
d	∞

From the reference trace of the program, we can accurately calculate the miss-rate by constructing a histogram of stack distances (shown with s). At a particular cache size of C, the cache already holds the item that was accessed within a stack distance of less than C (i.e., cache hits are the sum of requests with $s \leq C$). However, when an item is requested when more than C items were accessed before it, then it is a cache miss (i.e., cache misses are the sum of requests with $s > C$). Using these terms we can calculate the cache miss rate as follows:

$$m(C) = \frac{misses}{all} = \frac{\sum_{C+1 \leq i \leq \infty} s(i)}{\sum_{0 \leq i \leq \infty} s(i)} \tag{2.16}$$

While using stack distance provides an accurate modeling of the cache, it is a computationally expensive method as it needs analyzing request logs one-by-one, and requires $O(NlogM)$ time and $O(M)$ space for a trace of N accesses to M distinct elements [75].

- **Iterative Approach:** Dan and Towsley [35] derived an iterative algorithm under FIFO (i.e., the object replaced is the one that has been in the cache the longest) and LRU replacement policies. This iterative algorithm calculates the hit rate of a cache of size C using the hit rates for a cache of size $C - 1$. While accurate, the complexity and computational overhead of this approach is $O(CN)$ which is prohibitive in our case that N and C are very large.

Using Che's Approximation: Among the mentioned cache approximation techniques, we chose to model the cache with Che's approximation for the following reasons: (1) Despite providing an accurate solution, it is computationally less expensive than an iterative solution or the stack distance solution that needs replaying the request logs one by one (Our dataset has billions of log lines, See Section 2.6). (2) Che's approximation is a versatile and highly accurate tool for predicting the cache miss rate of a cache with LRU replacement, previous work [43] discuss in detail why the approximation works so well in different environments. In addition, we have verified the accuracy of Che's approximation against the iterative approach under different workloads in our environment. (3) Che's approximation does not make assumptions about the popularity distribution of libraries, allowing us to provide the $q(.)$ law to model different CPs. Using big-data frameworks such as Spark we can efficiently batch process the request logs of each CP to calculate the popularity distribution.

To use Che's approximation, we have analyzed the request logs of the top 1000 CPs in our dataset—these CPs make up more than 95% of the traffic in our dataset. Although the iterative approach is expensive, we have done extensive numerical verification of Che's approximation for each of these CPs against the iterative approach to ensure the Che's approximation is accurate. Still, estimating cache miss rates using Che's approximation for varying amounts of disk size is computationally expensive because it requires us to solve Equation 2.14 for each disk size to find the characteristic time of the cache; to avoid recalculating the root of this equation and speeding up the optimization, we pre-compute the cache miss rate across a wide range of disk sizes for each CP and then fit a curve to the model to obtain an analytical expression of $m_{i,j}$ with respect of $d_{i,j}$.

We also verified these curves of cache miss-rate vs. disk are convex curves via calculating the second derivative of curves, in addition to numerically verifying them. We can see the relationship between $m_{i,j}$ and $d_{i,j}$ of the same CPs in Figure 2.8.

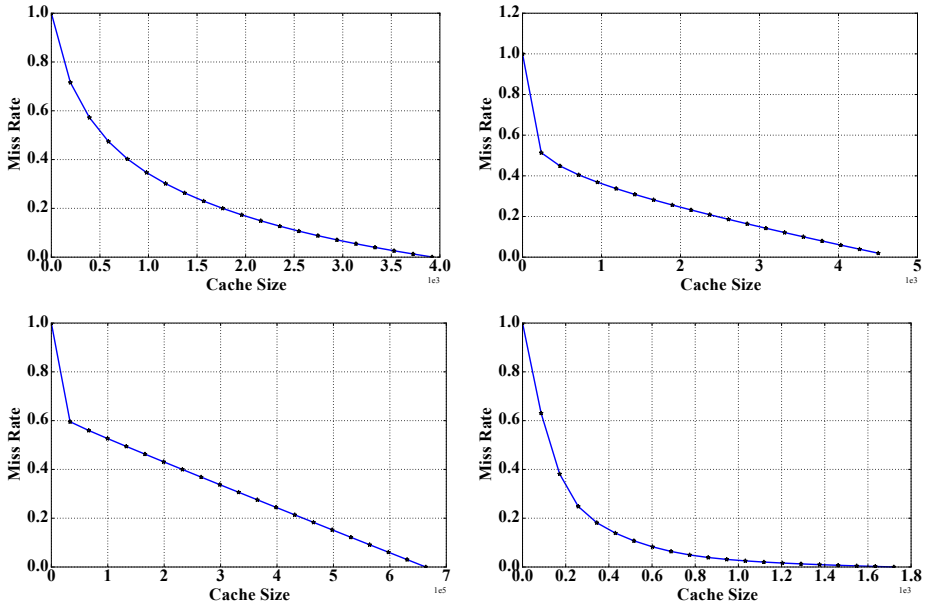


Figure 2.8: Cache miss-rate vs. disk size for some example CPs. We can see the convexity in curves.

2.6 Dataset

Our dataset contains two days worth of data, where we instrumented over 300 servers located across the state of Pennsylvania over two separate periods of 24 hours. The reason we chose the state of Pennsylvania for analysis is because Akamai has a relatively high number of edge servers located in this state, and it is a geographically broad enough state to have some differences in network latency or distances. We analyzed more than 4.5 billion request logs. This dataset includes which clients contacted which edge servers, to request the items belonging to a CP. Hence it allows us to construct the input to the optimization framework as shown in Table 2.1, in addition to calculating the per-CP popularity distribution analysis to use in Che's approximation.

Data preparation: Using our dataset, we first construct the path cost values between clients to edge servers and the edge servers to origin servers (i.e., the $a_{j,k}$ and $b_{i,j}$). These costs can either be the geographical distance, which we obtain via Aka-

mai's internal geo-location package called Edgescape [39], or the network latency (as in, the latency in retrieving the first-byte as observed by the clients, including both the server and the network latency), which is measured by the edge servers. We group the logs by CP to calculate the per-CP demand that is served from each edge server ($l_{i,j}$). We construct the popularity distribution per CP ($q(.)$); Next, we fit the cache miss rate vs. disk curves for each CP, according to the model explained in Section 2.5.

2.7 Results of Disk Optimization

Here we present the result of our optimization framework (Equations 2.4 to 2.11) on one edge server. Notice that each server runs the disk partitioning optimization locally, as opposed to the mapping and placement problems that are solved globally. Since the problem is a convex optimization, we use an off-the-shelf convex optimizer, the `fmincon` by Matlab [68], however note that the local minimum is guaranteed to be the global minimum. We chose one edge server to evaluate our methods on it, and we compare our solution against a shared LRU disk (i.e., non-managed LRU disk). This edge server is located in the state of PA. We will focus on the top 50 CPs of this server. We have analyzed the object sizes served from this server on this day, and observed that both the median and mean object sizes are around 1 MB; thus for simplicity we assume a uniform object size of 1 MB. However, note that Che's model can easily be extended to include different object sizes within a library [43]. For each CP, we calculate the popularity distributions and cost vectors to the origin servers. The disk size of this server is 1TB, thus it can fit 1 million average-sized objects in the cache.

Table 2.3 shows the preliminary results of our disk partitioning algorithm as compared to a non-managed disk where CPs share an LRU cache. The shared LRU cache is modeled as a single library that contains the items of all 50 CPs (i.e., constructing

$q(.)$ across all objects of all CPs). We can observe that for a slight increase (0.25%) in the overall cache miss rate (i.e., cost), network latency enhanced by 17% (i.e., performance) if the disk is partitioned according to different CPs's demand, popularity distribution, and latency to their origin. We can thus conclude that even with the current mapping and placement algorithms in place, the CDNs can enhance the performance by only managing the disk. Note that this post-mapping disk optimization is decentralized (i.e., optimization is run on each edge server as opposed to the entire platform), which makes the convex optimization simpler, and enables incremental deployment on the platform.

Table 2.3: Results of post-mapping disk optimization

Method	**Cache Miss Rate (cost)**	**Avg Distance on Miss (perf)**	**Avg Latency on Miss (perf)**
Shared LRU cache	22.50%	314 km	37.13 ms
Partitioned cache	22.75%	260 km	31.63 ms

2.8 Discussion and Limitations

There are some limitations in our current framework. Here we breifly describe them:

Object Sizes: So far we have assumed the cache capacity is measured in objects (i.e., unit object size of 1 MB). In reality, objects have different sizes and cache capacity is more reasonably measured in bytes. Fortunately, this simplification can be addressed in two ways: First, by using a version of Che's model that includes object sizes: suppose object n has size $\theta(n)$. Since the cache is intended to store a very large number of objects, we can assume that $\theta(n) << C$. This enables us to reasonably ignore boundary effects and adapt the Che approximation by replacing Equation 2.14 with:

$$\sum_{0 \leq n \leq N} (1 - e^{-q(n)t})\theta(n) = C \tag{2.17}$$

The second way to account for variable size objects is to assume they are divided into constant-sized chunks. Let $\theta(n)$ be given in chunks and assume all chunks inherit the popularity $q(n)$ of their parent object. Applying Che's approximation to chunks, it is easy to see that Equation 2.14 for the characteristic time is the same as Equation 2.17.

Variable Load and Popularity: Our framework is an offline algorithm that takes into account the popularity of items and the demand of each CP as an input. But if these inputs change, the result of optimization is no longer the valid solution. To deal with this issue, we propose an online algorithm as part of our future work; our online solution tracks changes in demands and popularity distribution per-CP, and recalculates the allocated disk amounts. Still, note that the popularity distribution of the library, $q(.)$, can stay the same while the underlying items change within the ranks (e.g., assume a CP introduces a new episode of a popular series every weekend, while the most popular item of the library will change, the values of $q(1)$ compared to $q(2)$ can stay similar, resulting in similar disk quota optimization). In addition, we can reserve a portion of disk on edge servers as a cushion to handle sudden changes in traffic such as flash crowds.

Caching Hierarchy: Our model of a CDN in this work ignores the caching hierarchy; but in reality CDNs have layers of edge servers and an edge server often consults their peers and parents before reaching the CP's origin servers. Fortunately, our method can be extended to include the caching hierarchy. To do this, we first need to estimate the popularity distribution of the forwarded traffic—the traffic that is forwarded to peers. Since we know the per-object miss rates and popularity distribution of each

CP, we can estimate the popularity distribution and demand of the forward traffic and repeat the disk optimization at every level of the hierarchy.

2.9 Related Work

There is a large body of related work, below we categorized them into relevant groups.

Cache Replacement Policy

There has been an extensive amount of work done on the cache replacement policy. In particular, [103] describes the elements of web caching systems (including hierarchical architecture, pre-fetching, etc.), and [79] offers an extensive survey of cache replacement policies. [108] identifies the appropriate policies for proxies with different characteristics, [28] aims to find the cache replacement policy (e.g., LRU vs LFU) best suited for video workloads, [71] proposes using a rank-based algorithm using the important characteristics of the video.

Our work differs from related work in this category because we try to optimize the resource allocation to reduce the impact of cache miss rates *given* the current replacement policy (LRU in our model).

Replica Selection Algorithms

There is a wide range of related work in this area [107, 85, 29, 89, 7, 50, 22, 82], and we direct the readers to consider surveys such as [102] for an overview of the work done so far. In particular, [67] overviews the current mapping system at Akamai including GLB and stable allocations, and [29] explains how the mapping benefits from end-user mapping instead of name servers.

Our work differs from the related work in several aspects: First, we characterize the *actual workloads* and allocate resources based on the real data. Second, we model a joint optimization that *explicitly models the impact of cache misses*, based on the results of our characterization. Third, we propose a mechanism to manage cache

miss rates by controlling disk sizes, instead of sharing a cache. Finally, we propose a unified framework for CDNs to solve resource allocation problems at once.

Handling Per-object Cache Misses

Hyperbolic caching [13] is a closely related work, in terms of incorporating different latencies to the origins in a different form than our partitioning the cache. The hyperbolic caching method decays item priorities at variable rates and continuously reorders many items at once.

Workload Characterization Our work differs from this category in two respects: first, unlike previous work [17, 81], we showed a lot of workloads do *not* follow the Zipfian distribution. Second, we use the characterized workloads's popularity curves in our optimization to allocate the resources effectively across the platform.

Cache Modeling

Che's approximation [27] proposes a simple approach for estimating the hit rates of an LRU cache. This approximation has been widely adopted, and [43] offers mathematical explanation for the success of Che's approximation. We also discussed an iterative approach to model the LRU cache [35]. LRU cache can also be modeled using stack distance, as first defined by [69].

Most of the existing work in this category model a *shared* cache for objects of a single library, as opposed to our solution that aims to *partition* the cache among several libraries. Still, our optimization uses an existing model (the Che's approximation) to construct an analytical model for cache miss vs. disk for different workloads.

Chapter 3

Diagnosis of Internet Video Anomalies

3.1 Introduction

In the previous chapter, we presented a framework to allocate resources efficiently across the CDN platform to strike a balance between performance and cost. We used a relatively simple model of performance based on the distance and network latency between the client and edge server, or the edge server and the origin server. But, end-to-end performance depends on a variety of other factors, including performance problems at the edge server, congestion in the network, or problems at the client machine. To uncover this wide range of problems at the end-to-end path, we present Diva in this chapter. Our goal here is to diagnose the roots of poor performance, focusing on one content provider. In this case, we focus on a video provider, since video is now the dominant application of the Internet.

Internet users watch hundreds of millions of videos per day [114], and video streams represent more than 70% of North America's downstream traffic during peak hours [86]. A video streaming session, however, may suffer from problems such as

long startup delay, re-buffering events, and low video quality that negatively impact user experience and the content provider's revenue [63, 37]. Content providers strive to improve performance through a variety of optimizations, such as placing servers closer to clients, content caching, effective peering and routing decisions, and splitting the video session (i.e., the HTTP session carrying the video traffic) into fixed-length *chunks* in multiple bitrates [3, 113, 55, 59, 98]. Multiple bitrates enable adaptive bitrate algorithms (ABR) in the player to adjust video quality to available resources.

Despite these optimizations, performance problems can arise anywhere along the end-to-end delivery path shown in Figure 3.1. The poor performance can stem from a variety of root causes. For example, the *backend service* may increase the chunk download latency on a cache miss. The *CDN* servers can introduce high latency when accessing data from disk. The *network* can introduce congestion or random packet losses. The *client's download stack* may handle data inefficiently (e.g., slow copying of data from OS to the player via the browser and Flash runtime) and the *client's rendering path* may drop frames due to high CPU load.

While ABR algorithms can adapt to performance problems (e.g., lower the bitrate when throughput is low), understanding the *location* and *root causes* of performance problems enables content providers to take the right corrective (or even proactive) actions, such as directing client requests to different servers, adopting a different cache-replacement algorithm, or further optimizing the player software. In some cases, knowing the bottleneck can help the content provider decide *not* to act, because the root cause is beyond the provider's control—for example, it lies in the client's browser, operating system, or access link. The content provider could detect the existence of performance problems by collecting Quality of Experience (QoE) metrics at the player, but this does not go far enough to identify the underlying cause. In addition, the buffer at the player can (temporarily) mask underlying performance problems, leading to delays in detecting significant problems based solely on QoE metrics.

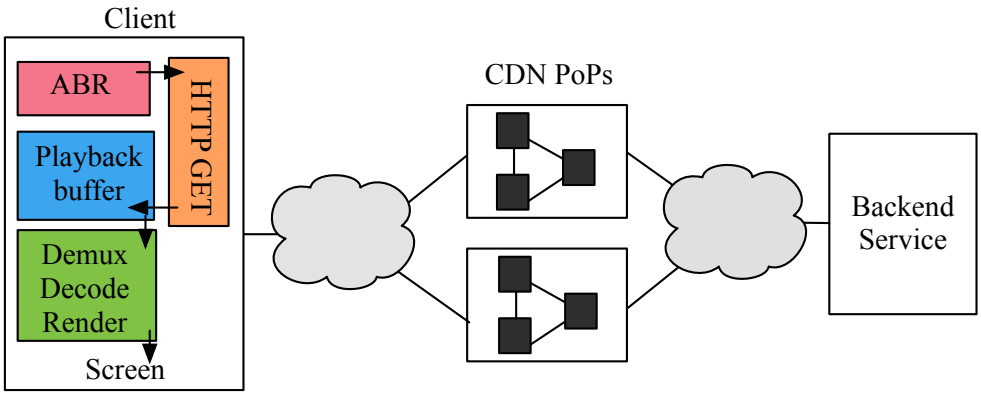


Figure 3.1: End-to-End video delivery components.

Instead, we adopt a *performance-driven* approach for uncovering performance problems. Collecting data at the client or the CDN alone is not enough. Client-side measurements, while crucial for uncovering problems in the download stack (e.g., a slow browser) or rendering path (e.g., slow decoder), cannot isolate network and provider-side bottlenecks. Moreover, a content provider cannot collect OS-level logs or measure the network stack at the client; even adding small extensions to the browsers or plugins would complicate deployment. Server-side logging can fill in the gaps [115], with care to ensure that the measurements are sufficiently lightweight in production.

In this paper, we instrument the CDN servers and the video player of a Web-scale commercial video streaming service, and join the measurement data to construct an end-to-end view of session performance. We measure per-*chunk* milestones at the player, which runs on top of Flash (e.g., the time to get the chunk's first and last bytes, and the number of dropped frames during rendering), and the CDN server (e.g., server and backend latency), as well as kernel-space TCP variables (e.g., congestion window and round-trip time) from the server host. Direct measurement of the main system components help us avoid relying on inference or tomography techniques that would limit the accuracy; or requiring other source of "ground truth" to label the data for machine learning [36]. We make the following contributions in this chapter:

Location	Findings
CDN	1. Asynchronous disk reads increase server-side delay.
	2. Cache misses increase CDN latency by order of magnitude.
	3. Persistent cache-miss and slow reads for unpopular videos.
	4. Higher server latency even on lightly loaded machines.
Network	1. Persistent delay due to physical distance or enterprise paths.
	2. Higher latency variation for users in enterprise networks.
	3. Packet losses early in a session have a bigger impact.
	4. Bad performance caused more by throughput than latency.
Client	1. Buffering in client download stack can cause re-buffering.
	2. First chunk of a session has higher download stack latency.
	3. Less popular browsers drop more frames while rendering.
	4. Avoiding frame drops needs min of $1.5\frac{sec}{sec}$ download rate.
	5. Videos at lower bitrates have *more* dropped frames.

Table 3.1: Summary of key findings.

1. An End-to-end instrumentation to diagnose performance problems within the player, the network path, and the CDN, across multiple layers of the stack, *per-chunk*. We show an example of how partial instrumentation (e.g., player-side alone) would lead to incorrect conclusions about performance problems. Such conclusions could cause the ABR algorithm to make wrong decisions.

2. A large-scale instrumentation of *both sides* of the video delivery path in a commercial video streaming service over a two-week period, studying more than 523 million chunks and 65 million on-demand video sessions.

3. We characterize transient and persistent problems in the end-to-end path that have not been studied before; in particular, the client's *download stack* and *rendering path*, and show their impact on QoE.

4. We offer a *comprehensive characterization* of performance problems for Internet video, and our key findings are listed in Table 3.1. Based on these findings, we offer insights for video content providers and Internet providers to improve video QoE.

3.2 Chunk Performance Monitoring

Model. We model a video session as an ordered sequence of HTTP(S)[1] requests and responses over a single TCP connection between the player and the CDN server—after the player has been assigned to a server. The session starts with the player requesting the manifest, which contains a list of chunks in available bitrates (upon errors and user events such as seeks, manifest is requested again). The ABR algorithm — tuned and tested in the wild to balance between low startup delay, low re-buffering rate, high quality and smoothness — chooses a bitrate for each chunk to be requested from the CDN server. The CDN service maintains a FIFO queue of arrived requests and maintains a thread pool to serve the queue. The CDN uses a multi-level distributed cache (between machines, and the main memory and disk on each machine) to cache chunks with an LRU replacement policy. Upon a cache miss, the CDN server makes a corresponding request to the backend service.

The client host includes two independent execution paths that share host resources. The *download* path "moves" chunks from the NIC to the player, by writing them to the playback buffer. The *rendering* path reads from the playback buffer, demuxes (audio from video), decodes and renders the pixels on the screen—this path could use either the GPU or the CPU. Note that there is a stack below the player: the player executes on top of a Javascript and Flash runtime, which in turn is run by the browser on top of the OS.

3.2.1 Chunk Instrumentation

We collect *chunk*-level measurements because: (1) most decisions affecting performance are taken per-chunk (e.g., caching at the CDN, and bitrate selection at the player), although some metrics are chosen once per session (e.g., the CDN server),

[1]Both HTTP and HTTPS protocols are supported at Yahoo; for simplicity, we use HTTP instead of HTTPS in the rest of the paper.

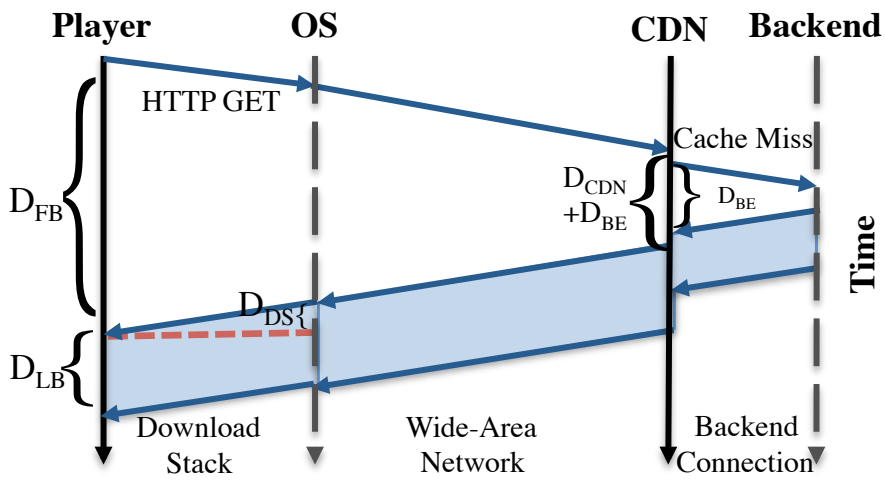


Figure 3.2: Time diagram of chunk delivery. Solid lines are instrumentation while dashed lines are estimates.

(2) sub-chunk measurements would increase CPU load on client, at the expense of rendering performance (Section 3.4.4), and (3) client-side handling of data within a chunk can vary across streaming technologies, and is often neither visible nor controllable. For example, players implemented on top of Flash use a progress event that delivers data to the player, and the buffer size or frequency of this event may vary across browsers or versions.

We capture the following milestones per chunk at the player and the CDN service: (1) When the chunk's HTTP GET request is sent, (2) CDN latency in serving the chunk, in addition to backend latency for cache misses, and (3) the time to download the first and last bytes of the chunk. We denote the player-side *first-byte delay* D_{FB} and *last-byte delay* D_{LB}. Figure 3.2 summarizes our notation. We divide a chunk's lifetime into the three phases: fetch, download, and playout.

Fetch Phase. The fetch process starts with the player sending an HTTP request to the CDN for a chunk at a specified bitrate until the first byte arrives at the player. The byte transmission and delivery traverse the host stack (player, Flash runtime, browser, userspace to kernel space and the NIC)—contributing to the *download stack*

latency. If the content is cached at the CDN server, the first byte is sent after a delay of D_{CDN} (the cache lookup and load delay); otherwise, the backend request for that chunk incurs an additional delay of D_{BE}. Note that the backend and delivery are always pipelined. The first-byte delay D_{FB} includes network round-trip time (rtt_0), CDN service latency, backend latency (if any), and client download stack latency:

$$D_{FB} = D_{CDN} + D_{BE} + D_{DS} + rtt_0 \tag{3.1}$$

We measure D_{FB} for each chunk at the player. At the CDN service, we measure D_{CDN} and its constituent parts: (1) D_{wait}: the time the HTTP request waits in the queue until the request headers are read by the server, (2) D_{open}: after the request headers are read until the server first attempts to open the file, regardless of cache status, and (3) D_{read}: time to read the chunk's first byte and write it to the socket, including the delay to read from local disk or backend. The backend latency (D_{BE}) is measured at the CDN service and includes network delay. Characterizing backend service problems is out of scope for this work; we found that such problems are relatively rare.

A key limitation of player-side instrumentation is that application layer metrics capture the mix of download stack latency, network latency, and server-side latency. To isolate network performance from end-host performance, we measure the end-to-end network path at the CDN host kernel's TCP stack. Since kernel-space latencies are relatively very low, it is reasonable to consider this view as representative of the network path performance. Specifically, the CDN service snapshots the Linux kernel's `tcp_info` structure for the player TCP connection (along with context of the chunk being served). The structure includes TCP state such as smoothed RTT, RTT variability, retransmission counts, and sender congestion window. We sample

the path performance periodically every 500ms[2]; this allows us to observe changes in path performance.

Download Phase. The download phase is the window between arrivals of the first and the last bytes of the chunk at the player, i.e., the last-byte delay, D_{LB}. It depends on the chunk size, which depends on chunk bitrate and duration. To identify chunks suffering from low throughput, on the client side we record the requested *bitrate* and the *last-byte delay*. To understand the network path performance and its impact on TCP, we snapshot TCP variables from the CDN host kernel at least once per-chunk (as described above).

Playout Phase. As a chunk is downloaded, it is added to the playback buffer. If the playback buffer does not contain enough data, the player pauses and waits for sufficient data; in case of an already playing video, this causes a rebuffering event. We instrument the player to measure the number (buf_{count}) and duration of rebuffering events (buf_{dur}) per-chunk played.

Each chunk must be decoded and rendered at the client. In the absence of hardware rendering (i.e., GPU), chunk frames are decoded and rendered by the CPU, which makes video quality sensitive to CPU utilization. A slow rendering process drops frames to keep up with the encoded frame rate. To characterize rendering path problems, we instrument the Flash player to collect the average rendered frame rate per chunk (avg_{fr}) and the number of dropped frames per chunk ($drop_{fr}$). A low rendering rate, however, is not always indicative of bad performance; for example, when the player is in a hidden tab or a minimized window, video frames are dropped to reduce CPU load [37]. To identify these scenarios, the player collects a variable (vis) that records if the player is visible when the chunk is displayed. Table 3.2 summarizes the metrics collected for each chunk at the player and CDN.

[2]The frequency is chosen to keep overhead low in production.

Location	Statistics
Player (Delivery)	sessionID, chunkID, D_{FB}, D_{LB}, bitrate
Player (Rendering)	buf_{dur}, buf_{count}, vis, $avg_{fr}, drop_{fr}$
CDN (App layer)	sessionID, chunkID, D_{CDN} (wait, open, and read), D_{BE}, cache status, chunk size
CDN (TCP layer)	CWND, SRTT, SRTTVAR, retx, MSS

Table 3.2: Per-chunk instrumentation at player and CDN.

3.2.2 Per-session Instrumentation

In addition to per-chunk milestones, we collect session metadata; see Table 3.3. A key to end-to-end analysis is to *trace* session performance from the player through the CDN (at the granularity of chunks). We implement tracing by using a globally unique session ID and per-session chunk IDs.

Location	Statistics
Player	sessionID, user IP, user agent, video length
CDN	sessionID, user IP, user agent, CDN PoP, CDN server, AS, ISP, connection type, location

Table 3.3: Per-session instrumentation at player and CDN.

Latency	Description
D_{FB}	Time to fetch the first byte
D_{LB}	Time to download the chunk (first to last byte)
D_{CDN}	CDN latency ($= D_{wait} + D_{open} + D_{read}$)
D_{BE}	Backend latency in cache miss
D_{DS}	Client's download stack latency
rtt_0	Network round-trip time during the first-byte exchange

Table 3.4: Latency notations and their description

3.3 Measurement Dataset

We study 65 million VoD sessions (523m chunks) with Yahoo, collected over a period of 18 days in September 2015. These sessions were served by a random subset of 85 CDN servers across the US. Our dataset predominantly consists of clients in North America (over 93%).

Figure 3.3a shows the cumulative distribution of the length of the videos. All chunks in our dataset contain six seconds of video (except, perhaps, the last chunk).

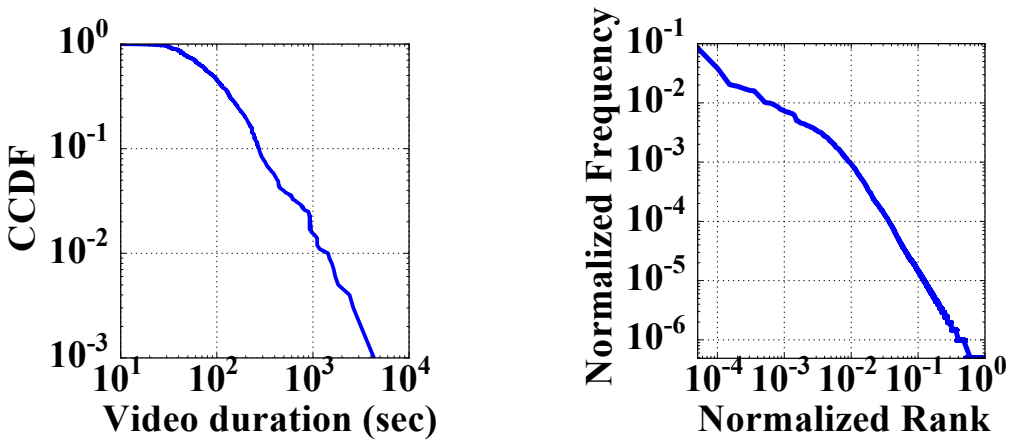
We focus on desktop and laptop sessions with Flash-based players. The browser distribution is as follows: 43% Chrome, 37% Firefox, 13% Internet Explorer, 6% Safari, and about 2% other browsers; the two major OS distributions in the data are Windows (88.5% of sessions) and OS X (9.38%). We do not consider cellular users in this paper since the presence of ISP proxies affects the accuracy of our findings.

The video viewership and popularity of videos is heavily skewed towards popular content; see Figure 3.3b. We find that top 10% of most popular videos receive about 66% of all playbacks.

Data preprocessing to filter proxies. A possible pitfall in our analysis is the existence of enterprise or ISP HTTP proxies [111], since the CDN server's TCP connection would terminate at the proxy, leading to network measurements (e.g., RTT) reflecting the server-proxy path instead of the client. We filter sessions using a proxy when: (i) we see different client IP addresses or user agents [104] between HTTP requests and client-side beacons[3], or (ii) the client IP address appears in a very large number of sessions (e.g., more more minutes of video per day than there are minutes in a day). After filtering proxies, our dataset consists of 77% of sessions.

Ethical considerations: Our instrumentation methodology is based on logs/metrics about the traffic, without looking at packet payload or video content. For privacy reasons, we do not track users (through logging) hence we cannot study access pat-

[3]A beacon is a message sent back from the client to the analytic servers, carrying information.



(a) CCDF of video lengths (one month) (b) Rank vs. popularity (one day)

Figure 3.3: Length and popularity of videos in the dataset.

terns of individual users. Our analysis uses client IP addresses internally to identify proxies and perform coarse-grained geo-location; after that, we use opaque session IDs to study the dataset.

3.4 Characterizing Performance

In this section, we characterize the performance of each component of the end-to-end path, and show the impact on QoE. Prior work has shown that important stream-related factors affect the QoE: startup delay, rebuffering ratio, video quality (average bitrate), and the rendering quality [37, 113]. They have developed models for estimating QoE scores of videos by assigning weights to each of these stream metrics to estimate a user behavior metric such as abandonment rate.

We favor looking at the impact on individual QoE factors instead of a single QoE score to assess the significance of performance problems. This is primarily because of the impact of content on user behavior (and hence, QoE). First, user behavior may be different for long-duration content such as Netflix videos (e.g., users may be more *patient* with a longer startup delay) than short-duration content (our case). Second, the type of content being viewed impacts user behavior (and hence the weights of QoE

factors). For example, the startup delay for a news video (e.g., "breaking news") may be more important to users than the stream quality; while for sports videos, the quality may be very important. Given the variety of Yahoo videos, we cannot use a one-size-fits-all set of weights for a QoE model. Moreover, the results would not generalize to all Internet videos. Instead, we show the impact of each problem directly on the QoE factors.

3.4.1 Server-side Performance Problems

Yahoo uses the Apache Traffic Server (ATS), a popular caching proxy server [5], to serve HTTP requests. The traffic engineering system maps clients to CDN nodes using a function of geography, latency, load, cache likelihood, etc. In other words, the system tries to route clients to a server that is likely to have a hot cache. The server first checks the main memory cache, then tries the disk, and finally sends a request to a backend server if needed.

Server latencies are relatively low, since the CDN and the backend are well-provisioned. About 5% of sessions, however, experience a QoE problem due to the server, and the problems can be *persistent* as we show below. Figure 3.4 shows the impact of the server-side latency for the first chunk on the startup delay (time to play) at the player.

1. Asynchronous disk read timer and cache misses cause high server latency. Figure 3.5 shows the distribution of each component of CDN latency across chunks; it also includes the distribution of total server latency for chunks broken by cache hit and miss. Most of the chunks have a negligible waiting delay ($D_{wait} < 1ms$) and open delay. However, the D_{read} distribution has two nearly identical parts, separated by about 10ms. The root cause is that ATS executes an asynchronous read to read the requested files in the background. When the first attempt in opening the

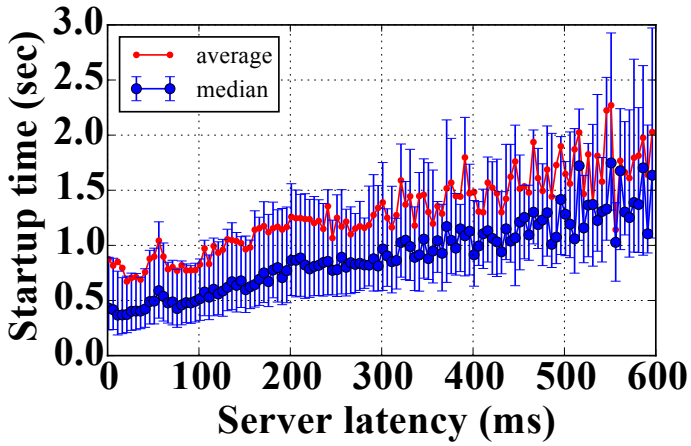


Figure 3.4: Impact of server latency on QoE (startup time), error bars show the interquartile range (IQR).

cache is not immediately returned (due to content not being in memory), ATS retries to open the file (either from the disk or from backend service) using a 10ms timer [99].

On a cache miss, the backend latency significantly affects the serving latency according to Figure 3.5. The median server latency among chunks experiencing a cache hit is 2ms, while the median server latency for cache misses is 40 times higher at 80ms. The average and 95^{th} percentile of server latency in case of cache misses increases tenfold. In addition, cache misses are the main contributor when server latency has a higher contribution to D_{FB} than the network RTT: for 95% of chunks, network latency is higher than server latency; however, among the remaining 5%, the cache miss ratio is 40%, compared to an average cache miss rate of 2% across session chunks.

Take-away: Cache misses impact serving latency, and hence QoE (e.g., startup time) significantly. An interesting direction to explore is to alter the LRU cache eviction policy to offer better cache hit rates. For example, policies for popular-heavy workloads, such as GD-size or perfect-LFU [17].

2. Less popular videos have persistent high cache miss rate and high latency. We observed that a small fraction of sessions experience performance prob-

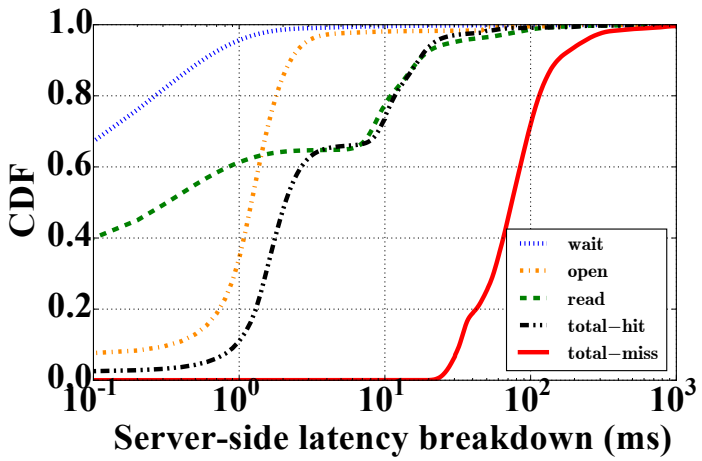


Figure 3.5: CDN latency breakdown across all chunks.

lems that are *persistent*. Once a session has a cache miss on one chunk, the chance of further cache misses increases significantly; the mean cache miss ratio among sessions with at least one cache miss is 60% (median of 67%). Also, once a session has at least one chunk with a high latency ($> 10ms$), the chance of future read delays increases; the mean ratio of high-latency chunks in sessions with at least one such chunk is 60% (median of 60%).

One possible cause for persistent latency, even when the cache hit ratio is high, is a highly loaded server that causes high serving latency; however, our analysis shows that server latency is not correlated with load[4]. This is because the CDN servers are well provisioned to handle the load.

Instead, the *unpopularity* of the content is a major cause of the persistent server-side problems. For less popular videos, the chunks often need to come from disk, or worse yet, the backend server. Figure 3.6(a) shows the cache miss percentage versus video rank (most popular video is ranked first) using data from one day. The cache miss ratio drastically increases for unpopular videos. Even on a cache hit, unpopular videos experience higher server delay, as shown in Figure 3.6(b). The figure shows mean server latency after removing cache misses (i.e., no backend communication).

[4]We estimated load as of number of parallel HTTP requests, sessions, or bytes served per second.

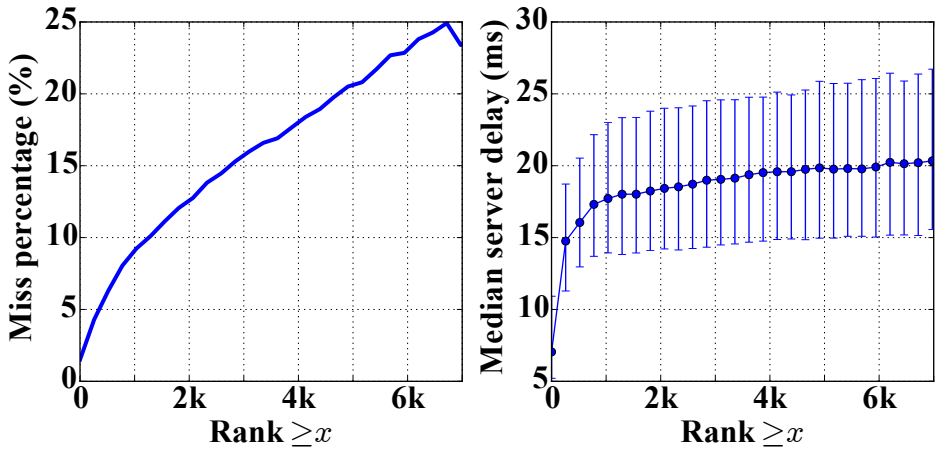


Figure 3.6: Performance vs popularity: (a) miss rate vs rank, (b) CDN latency (excluding cache misses) vs rank.

The unpopular content generally experiences a higher latency due to higher read (seek) latency from disk.

Take-away. The persistence of cache misses could be addressed by pre-fetching the subsequent chunks of a video session after the first miss. Pre-fetching of subsequent chunks would particularly help with unpopular videos since backend latency makes up a significant part of their overall latency and could be avoided.

When an object cannot be served from local cache, the request will be sent to the backend server. For a popular object, many concurrent requests may overwhelm the backend service; thus, the ATS retry timer is used to reduce the load on the backend servers; the timer introduces extra delay for cases where the content is available on local disk.

3. Load vs. performance due to cache-focused client mapping. We have observed that more heavily loaded servers offer *lower* CDN latency (note that CDN latency does not include the network latency, but only the time a server takes to start serving the file). This result was initially surprising since we expect busier servers to have worse performance; however, this can be explained by the *cache-focused mapping* CDN feature: As a result of cache-based assignment of clients to CDN servers, servers

with less popular content have more chunks with either higher read latency as the content is not fresh in memory (and the ATS retry-timer), or worse yet, need to be requested from backend due to cache-misses.

While unpopular content leads to lower performance, because of lower demand it also produces fewer requests, hence servers that serve less popular content seem to have worse performance at a lower load than the servers with a higher load.

Take-away. An interesting direction to achieve better utilization of servers and load balancing is to actively partition popular content among servers (on top of cache-focused routing). For example, given that the top 10% of videos make up 66% of requests, distributing only the top 10% of popular videos across servers can balance the load.

3.4.2 Network Performance Problems

Network problems can manifest themselves in the form of increased packet loss, re-ordering, high latency, high variation in latency, and low throughput. Each can be persistent (e.g., far away clients from a server have persistent high latency) or transient (e.g., spike in latency caused by congestion). In this section, we characterize these problems.

Distinguishing between a transient and a persistent problem matters because although a good ABR may *adapt* to temporary problems (e.g., by lowering bitrate), it cannot avoid bad quality caused by persistent problems (e.g., when a peering point is heavily congested, even the lowest bitrate may see re-buffering). Instead, persistent problems require *corrective actions* taken by the video provider (e.g., placement of new CDN PoPs) or ISPs (e.g., additional peering).

We characterize the impact of loss and latency on QoE. To characterize long-term problems, we aggregate sessions into /24 IP prefixes since most allocated blocks and BGP prefixes are /24 prefixes [80, 42]. Figure 3.7 shows the effect of network latency

during the first chunk on video QoE, specifically, startup delay, across sessions. High latency in a session could be caused by a persistently high baseline (i.e., high $srtt_{min}$)[5], or variation in latency as a result of transient problems (i.e., high variation, σ_{srtt}). Figure 3.8 shows the distribution of both of these metrics across sessions. We see that both of these problems exist among sessions; we characterize each of these next.

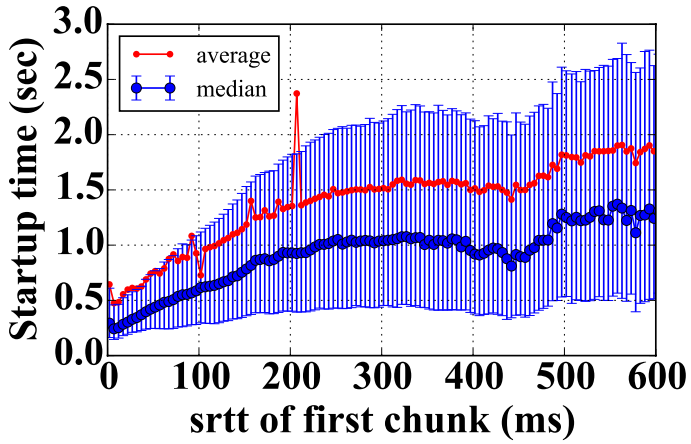


Figure 3.7: Average and median startup delay vs. network latency, error bars show the interquartile range (IQR).

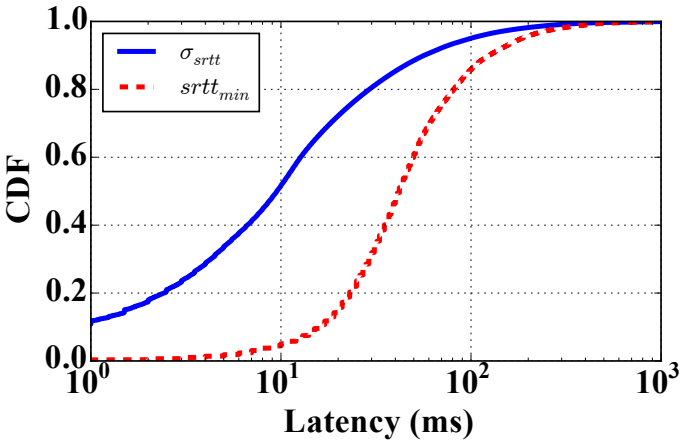


Figure 3.8: CDF of baseline ($srtt_{min}$) and variation in latency (σ_{srtt}) among sessions.

[5]Note that TCP's estimate of RTT, SRTT, is an EWMA average; hence $srtt_{min}$ is higher than the minimum RTT seen by TCP. The bias of this estimator, however, is not expected to be significant for our study since it is averaged.

1. Persistent high latency caused by distance or enterprise path problems. In Figure 3.8, we see that some sessions have a high minimum RTT. To analyze the minimum latency, it is important to note that the SRTT samples are taken after 500ms from the beginning of the chunk's transmission; hence, if a chunk has self-loading [57], the SRTT sample may reflect the additional queuing delay and not just the baseline latency. To filter out chunks whose SRTT has grown while downloading, we use an estimate of the initial network round-trip time (rtt_0) per-chunk. Equation 3.1 shows that $D_{FB} - (D_{CDN} + D_{BE})$ can be used as an upper-bound estimate of rtt_0. We take the minimum of SRTT and rtt_0 per-chunk as the baseline sample. Next, to find the minimum RTT in a session or prefix, we take the minimum among all these per-chunk baseline samples in the session or prefix.

In order to find the underlying cause of persistently high latency, we aggregate sessions into /24 client prefixes. The aggregation overcomes client last-mile problems, which may increase the latency for one session, but are not persistent problems. A prefix has more RTT samples than a session; hence, congestion is less likely to inflate all samples.

We focus our analysis on prefixes in the 90^{th} percentile latency, where $srtt_{min} > 100ms$; which is a high latency for cable/broadband connections (note that our CDN and client footprint is largely within North America). To ensure that a temporary congestion or routing change has not affected samples of a prefix, and to understand the persistent problems in poor prefixes, we repeat this analysis every day in our dataset and calculate the recurrence frequency, $\frac{\#days\ prefix\ in\ tail}{\#days}$. We take the top 10% of prefixes with highest re-occurrence frequency as prefixes with a persistent latency problem. This set includes $57k$ prefixes.

In these $57k$ prefixes, 75% are located outside the US and are spread across 96 different countries. These non-US clients are often limited by geographical distance and propagation delay. However, among the 25% of prefixes located in the US, *the*

majority are close to CDN nodes. Since IP geolocation packages may not be accurate outside US, in particular favoring the US with 45% of entries [80], we focus our geo-specific analysis to US clients. Figure 3.9 shows the relationship between the $srtt_{min}$ and geographical distance of these prefixes in the US. If a prefix is spread over several cities, we use the average of their distances to the CDN server. Among high-latency prefixes inside the US within a 4km distance, only about 10% are served by residential ISPs, while the remaining 90% of prefixes originate from corporations and private enterprises.

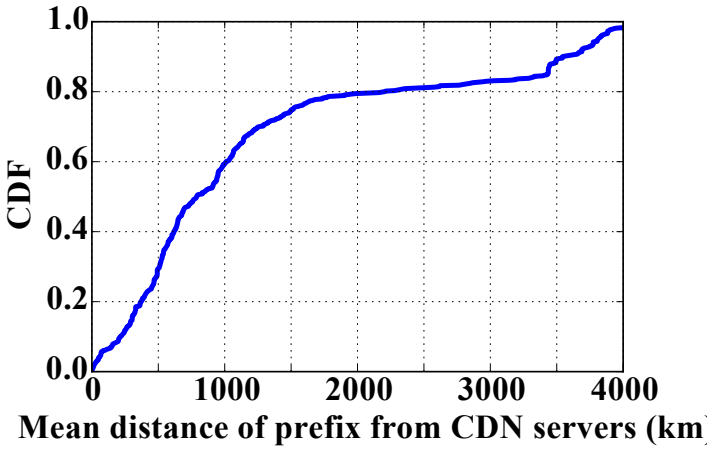


Figure 3.9: Mean distance (km) of US prefixes in the tail latency from CDN servers.

Take-away: Finding clients that suffer from persistent high latency due to geographical distance helps video content providers in better placement of new CDN servers and traffic engineering. It is equally important to look at close-by clients suffering from high latency to (1) avoid over-provisioning servers in those geographics and wasting resources, and, (2) identify the IP prefixes with known persistent problems and adjust the ABR algorithm accordingly, for example, to start the streaming with a more conservative initial bitrate.

2. Residential networks have lower latency variation than enterprises. To measure RTT variation, we calculate the coefficient of variation (CV) of SRTT in

each session, which is defined as the standard deviation over the mean of SRTT. Sessions with low variability have $CV < 1$ and sessions with high SRTT variability have $CV > 1$. For each ISP and organization, we measure the ratio of sessions with $CV > 1$ to all sessions. We limit the result to ISPs/organizations that have least 50 video streaming sessions to provide enough evidence of persistence. Table 3.5 shows the top ISPs/organizations with highest ratio. Enterprises networks make up most of the list. To compare this with residential ISPs, we analyzed five major residential ISPs and found that about 1% of sessions have $CV > 1$.

In addition to per-session variation in latency, we characterize the variation of latency in prefixes as shown in Figure 3.10. We use the average *srtt* of each session as the sample latency. To find the coefficient of variance among all source-destination paths, we group sessions based on their prefix and the CDN PoP. We see that 40% of (prefix, PoP) pairs belong to paths with high latency variation ($CV > 1$).

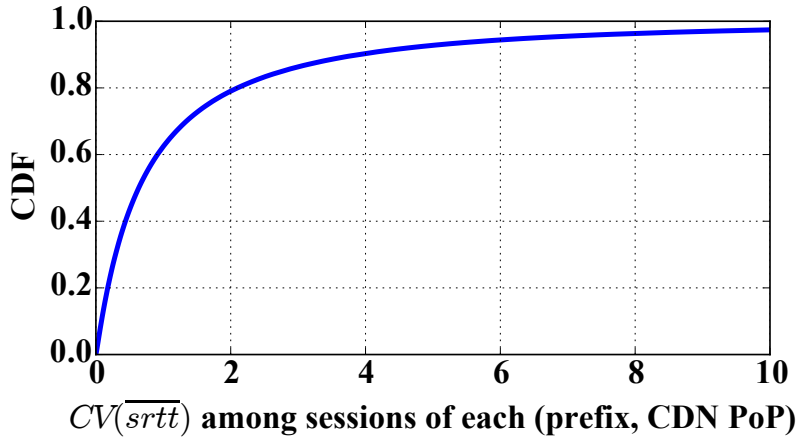


Figure 3.10: CDF of path latency variation: CV of latency per path, a path is defined by a (prefix, PoP) pair.

Take-away: Recognizing which clients are more likely to suffer from latency variation is valuable for content providers because it helps them make informed decisions about QoE. In particular, the player bitrate adaptation and CDN traffic engineering algorithms can use this information to optimize streaming quality under high latency

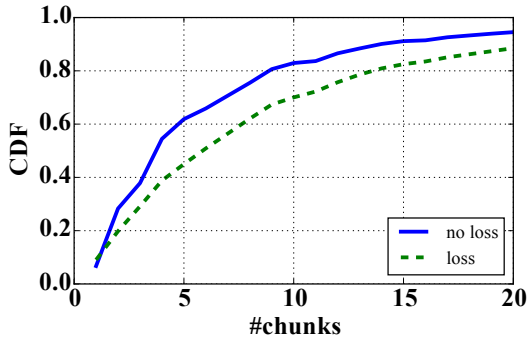
isp/organization	#sessions with $CV > 1$	#all sessions	Percentage
Enterprise#1	30	69	43.4%
Enterprise#2	4,836	11,731	41.2%
Enterprise#3	1,634	4,084	40.0%
Enterprise#4	83	208	39.9%
Enterprise#5	81	203	39.9%

Table 3.5: ISP/Organizations with highest percentage of sessions with $CV(SRTT) > 1$.

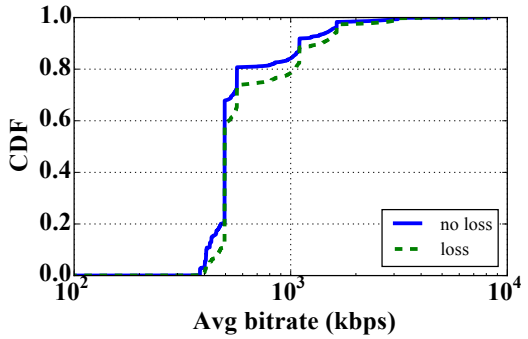
variation. For example, the player can make more conservative bitrate choices, lower the inter-chunk wait time (i.e., request chunks sooner), and increase the buffer size to deal with variability.

3. Earlier packet losses have higher impact on QoE. We use the retransmission count to study the effect of packet losses. A majority of the sessions ($> 90\%$) have a retransmission rate of less than 10%, with 40% of sessions experiencing no loss. While 10% can severely impact TCP throughput, not every retransmission is caused by an actual loss (e.g., due to early retransmit optimizations, underestimating RTO, etc.). Figure 3.11 shows the differences between sessions with and without loss in three aspects: (a) number of chunks (are these sessions shorter?), (b) bitrate (similar quality?), and (c) re-buffering. We see that the session length and bitrate distributions are almost similar between the two groups; however, re-buffering difference is significant and sessions without loss have better QoE.

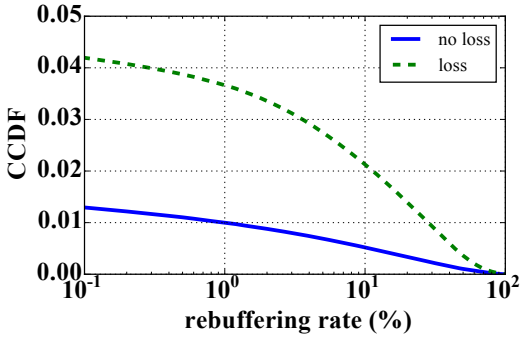
While higher loss rates generally indicate higher re-buffering (Figure 3.12), the loss rate of a TCP connection does not necessarily correlate with the video QoE; the timing of the loss matters too. Figure 3.13 shows two example sessions (case-1 and case-2) where both sessions have 10 chunks with similar bitrates, cache status, and SRTT distributions. Case-1 has a retransmission rate of 0.75% compared to 22% in case-2; but it experienced dropped frames and re-buffering despite the lower loss rate.



(a) CDF of session length with and without loss



(b) CDF of Average bitrate with and without loss



(c) CCDF (1-CDF) of Re-buffering rate with and without loss

Figure 3.11: Differences in session length, quality, and re-buffering with and without loss.

As Figure 3.13 shows, the majority of losses in case-1 happen in the first chunk, while case-2 has no loss during the first four chunks, building up its buffer to 29.8 seconds before a loss happens and successfully avoids re-buffering.

Because the buffer can hide the effect of subsequent loss, we believe that it is important to not only measure loss rate in video sessions, but also *the chunk ID that experiences loss*. Loss during earlier chunks has more impact on QoE because the playback buffer would hold less data for earlier chunks. We expect losses during the first chunk to have the highest effect on re-buffering. Figure 3.15 shows two examples: (1) $P(rebuf\ at\ chunk = X)$, which is the percentage of chunks with that chunk ID seeing a re-buffering event; and (2) $P(rebuf\ at\ chunk = X | loss\ at\ chunk = X)$, which is the same probability conditioned on occurrence of a loss during the chunk. While

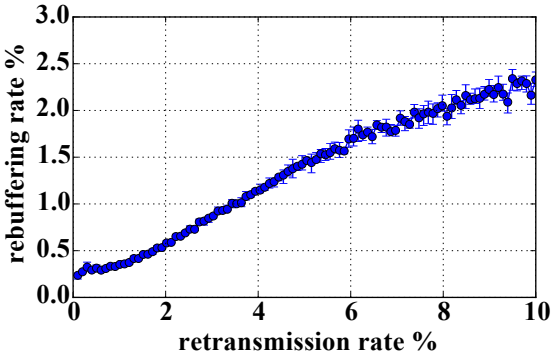


Figure 3.12: Rebuffering vs. retransmission rate in sessions.

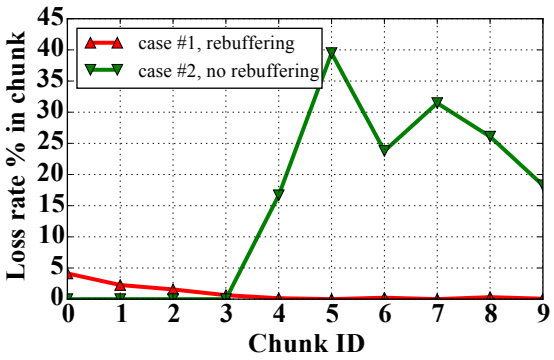


Figure 3.13: Example case for loss vs. QoE.

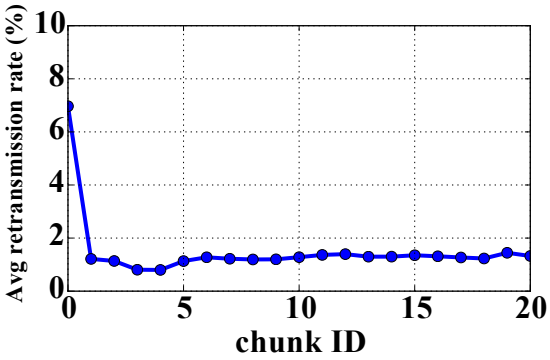


Figure 3.14: Average per-chunk retransmission rate.

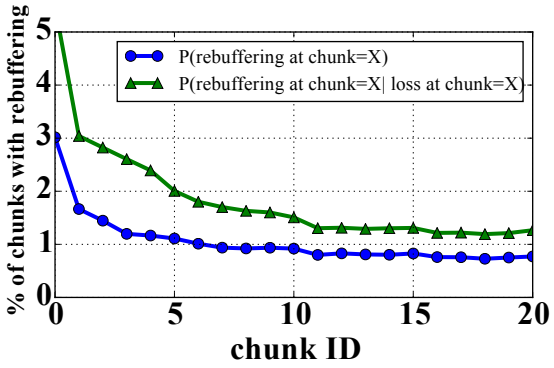


Figure 3.15: Re-bufffering frequency with or without loss, per chunkID.

occurrence of a loss in any chunk increases the likelihood of a re-buffering event, the increase is more significant for the first chunk.

We observe that losses are more likely to happen on the first chunk: Figure 3.14 shows the average per-chunk retransmission rate. The bursty nature of TCP losses

towards the end of slow start [1] could be the cause of higher loss rates during the first chunk, which TCP avoids in subsequent chunks when transitioning into congestion avoidance state.

Take-aways: Due to the existence of a buffer in video streaming clients, the session loss rate does not necessarily correlate with QoE. The temporal location of loss in the session matters as well: earlier losses impact QoE more, with the first chunk having the biggest impact.

Due to the bursty nature of packet losses in TCP slow start caused by the exponential growth, the first chunk may have the highest per-chunk retransmission rate. Prior work showed a possible solution to work around a related issue using server-side pacing [49].

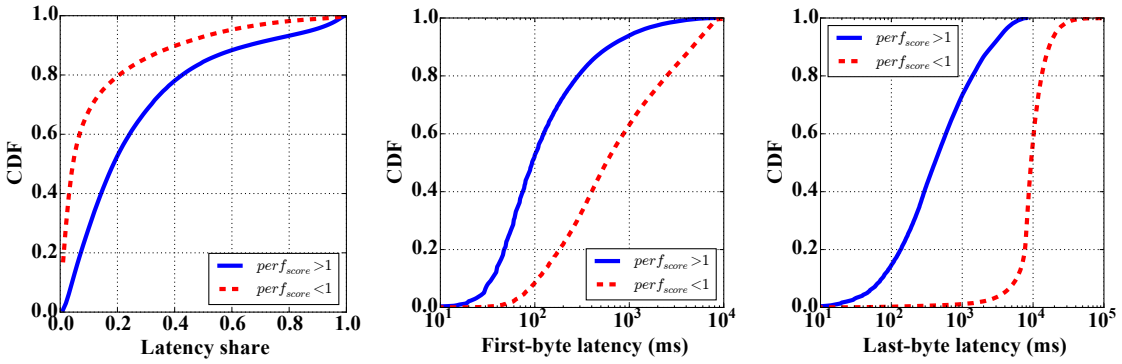
4. Throughput is a bigger problem than latency. To separate chunks based on performance, we use the following intuition: the playback buffer decreases when it takes longer to download a chunk than there are seconds of video in the chunk. With τ as the chunk duration, we tag chunks with bad performance when the following score is less than one:

$$perf_{score} = \frac{\tau}{D_{FB} + D_{LB}} \tag{3.2}$$

We use D_{LB} as a "measure" of throughput. Both latency (D_{FB}) and throughout (D_{LB}) play a role in this score. We define the latency share in performance by $\frac{D_{FB}}{D_{FB}+D_{LB}}$ and the throughput share by $\frac{D_{LB}}{D_{FB}+D_{LB}}$. We show that while the chunks with bad performance generally have higher latency and lower throughput than chunks with good performance, throughput is a more "dominant" metric in terms of impact on the performance of the chunk. Figure 3.16a shows that chunks with good performance generally have higher share of latency and lower share of throughput than chunks with bad performance. Figure 3.16b shows the difference in absolute values of D_{FB}, and Figure 3.16c shows the difference in absolute values of D_{LB}.

While chunks with bad performance generally have higher first and last byte delays, the difference in D_{FB} is negligible compared to that of D_{LB}. We can see that most chunks with bad performance are limited by throughout and have a higher throughput share.

Take-away: Our findings could be good news for ISPs because throughput can be an easier problem to fix (e.g., establish more peering points) than latency [64].



(a) Latency share ($\frac{D_{FB}}{D_{FB}+D_{LB}}$) (b) D_{FB} (c) D_{LB}

Figure 3.16: Latency vs throughput: (a) Latency share ($\frac{D_{FB}}{D_{FB}+D_{LB}}$), (b) D_{FB}, and (c) D_{LB} vs. performance score.

3.4.3 Client's Download Stack

1. Some chunks have significant download stack latency. Video packets traversing the client's download stack (OS, browser, and the Flash plugin) may be delayed due to buffered delivery. In the extreme case, all the chunk bytes could be buffered and delivered late and all at once to the player[6], resulting in a significant increase in D_{FB}. Since the buffered data is delivered at once or in short time windows, the *instantaneous throughput* ($TP_{inst} = \frac{chunk\ size}{D_{LB}}$) will be much higher at the player than the arrival rate of the chunk bytes from the network. We use TCP variables to

[6]Note that the delay is not caused by a full playback buffer, since the player will not request a new chunk when the buffer is full.

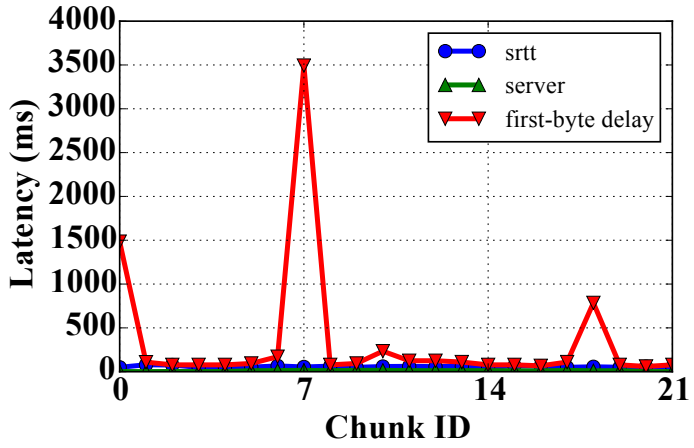
estimate the download throughout per-chunk:

$$throughput = MSS \times \frac{CWND}{SRTT} \tag{3.3}$$

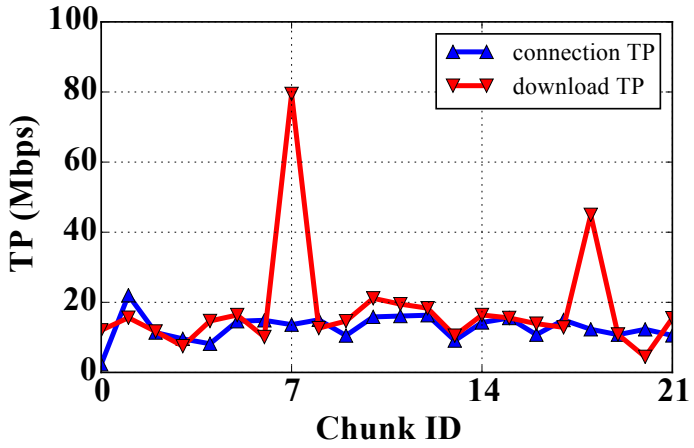
To detect chunks with this issue, we detect outliers using standard deviation: when a chunk is buffered in the download stack, its D_{FB} is much higher than that of the other chunks — more than $2 \cdot \sigma$ greater than the mean — despite other similar latency metrics (i.e., network and server-side latency are within one σ of the mean). Also, its TP_{inst} is much higher — more than $2 \cdot \sigma$ greater than the mean — due to the buffered data being delivered in a shorter time, while the estimated throughput from server side (using CWND and SRTT) does not explain the increase in throughput. Equations 3.4 summarize the detection conditions:

$$\begin{aligned} D_{FB_i} &> \mu_{D_{FB}} + 2 \cdot \sigma_{D_{FB}} \\ TP_{inst_i} &> \mu_{TP_{inst}} + 2 \cdot \sigma_{TP_{inst}} \\ SRTT, D_{server}, CWND &< \mu + \sigma \end{aligned} \tag{3.4}$$

Figure 3.17 shows an example session that experiences the download stack problem (DS) taken from our dataset; our algorithm detected chunk 7 with much higher D_{FB} and TP_{inst} than the mean. Figure 3.17a shows D_{FB} of chunks and its constituents parts. We see that the increase in chunk 7's D_{FB} is not caused by a latency increase in backend, CDN, or network RTT. Figure 3.17b shows that this chunk also has an abnormally high throughput that seems impossible based on the estimated network throughput (Equation 3.3) at the server-side. The presence of both observations in the same chunk suggests that the chunk was buffered inside the client's stack and delivered late to the player. The buffered data was delivered almost instantaneously, since it mainly involves a kernel to userspace copy.



(a) First-byte delay and its constituents in the session



(b) Network throughput vs instantaneous download throughput

Figure 3.17: A case study showing the effects of client download stack (chunk#7).

We have detected 1.7m chunks (0.32% of all chunks) using this method, demonstrating how often the client's download stack can buffer the data and hurt performance. About 1.6m video sessions have at least one such chunk (3.1% of sessions).

Take-aways: The download stack problem is an example where looking at one-side of measurements (CDN or client) alone would lead to wrong conclusions, and where both sides may blame the network. It is only with end-to-end instrumentation that this problem can be localized correctly. Failure in correctly recognizing such an effect on latency may lead to the following problems:

Over-shooting: Some ABR algorithms use player-level throughput (i.e., the instantanous throughput) in the bitrate selection process (e.g., a moving average of previous N chunks' throughput). Buffered delivery can lead to overestimation of end-to-end TCP throughput.

Under-shooting: If the ABR algorithms are either latency-sensitive, or use the average throughput (as opposed to the instantaneous throughput), the affected chunks may cause underestimation of the connection's throughput.

Incorrect actions: When the TCP throughput is low, content providers may initiate a corrective action, such as re-routing the client. If the download stack latency is not diagnosed, clients may be falsely re-routed.

While designing ABR algorithms that rely on throughput or latency measurements, using server-side data (CWND and SRTT) enables the player to estimate the state of the network more accurately than client-side measurements alone. This could be done by the CDN in an HTTP header with the next chunk. When it is not possible to incorporate server-side measurements, the current ABR algorithms that rely on client-side measurements should detect and exclude outliers in their throughput/latency input.

2. Persistent download-stack problems. The underlying assumption in the above method is that the majority of chunks will not be buffered by download stack, hence we can *detect the outlier* chunks. However, when a persistent problem in client's download stack affects all or most chunks, this method cannot detect the problem. If we could directly observe the network RTT, rtt_0, we can estimate D_{DS} using Equation 3.1 per-chunk. The current vanilla Linux kernel does not expose individual RTT samples via the `tcp_info` structure, and kernel changes or collecting packet traces may be infeasible in production settings.

To work around this limitation, we use a conservative estimate of rtt_0 as the TCP retransmission timer (RTO)[7]. *RTO* is how long the sender waits for a packet's acknowledgment before deciding it is lost; hence, RTO can be considered as a conservative estimate of rtt_0. We use RTO to estimate a lower bound of the client download stack latency per-chunk:

$$D_{DS} \geq D_{FB} - D_{CDN} - D_{BE} - RTO \tag{3.5}$$

Using this method, we see that 17.6% of all chunks experience a positive download stack latency. In 84% of these chunks, download stack latency share in D_{FB} is higher than network and server latencies, making it the bottleneck in D_{FB}. Table 3.6 shows the top OS/browser combinations with highest persistent download stack latency. We see that among major browsers, Safari on non-OS X environments has the highest average download stack latency. In the "other" category, we find that less-popular browsers on Windows, in particular, Yandex and SeaMonkey, have high download stack latencies.

Browser / OS	*Safari on Linux*	*Safari on Windows*	*Firefox on Windows*	*Other on Windows*	*Firefox on Mac*
mean DS(ms)	1041	1028	283	281	275

Table 3.6: OS/browser with highest D_{DS}.

Take-aways and QoE impact: Download stack problems are worse for sessions with re-buffering: among sessions with no re-buffering, the average D_{DS} is less than 100ms. In sessions with up to 10% re-buffering ratios, the average D_{DS} grows up to 250ms, and in sessions with more than 10% re-buffering ratios, the average D_{DS} is more than 500ms. Although the download stack latency is not a frequent problem, it is important to note that when it is an issue, it is often the major bottleneck

[7] $RTO = 200ms + SRTT + 4 \times SRTTVAR$, according to RFC 2988 [77].

in latency. Any adaptation mechanisms at the client should detect the outliers to improve QoE.

It is important to know that some client setups (e.g., Yandex or Safari on Windows) are more likely to have persistent download stack problems. Recognizing the lasting effect of client's machine on QoE helps content providers avoid actions caused by wrong diagnosis (e.g., re-routing clients due to *seemingly* high network latency when problem is in download stack).

3. First chunks have higher download stack latency. We find that the distribution of D_{FB} in first chunks is higher than other chunks: the median D_{FB} among first chunks is $300ms$ higher than other chunks. Using packet traces and developer tools on browsers, we confirmed that this effect is not visible in OS or browser timestamps. We believe that the difference is due to higher download stack latency of first chunk. To test our hypothesis, we select a set of *performance-equivalent* chunks with the following conditions: (1) no packet loss, (2) CWND > ICWND, (3) no queuing delay and similar SRTT (we use $60ms < SRTT < 65ms$ for presentation), and (4) $D_{CDN} < 5ms$, and cache-hit chunks.

Figure 3.18 shows the distribution of D_{FB} among the equivalent set for first versus other chunks. We see that despite similar performance conditions, first chunks experience higher D_{FB}. The root cause appears to be the processing time spent in initialization of Flash events and data path setup (using the progressEvent in Flash) at the player, which can increase D_{FB} of first chunk.[8]

Take-away: First chunks experience a higher latency than other chunks. Video providers could eliminate other sources of performance problems at startup and reduce the startup delay by methods such as caching the first chunk of video titles [87], or by assigning higher cache priorities for first chunks.

[8]We can only see Flash as a blackbox, hence, we cannot confirm this. However, a similar issue about ProgressEvent has been reported [41].

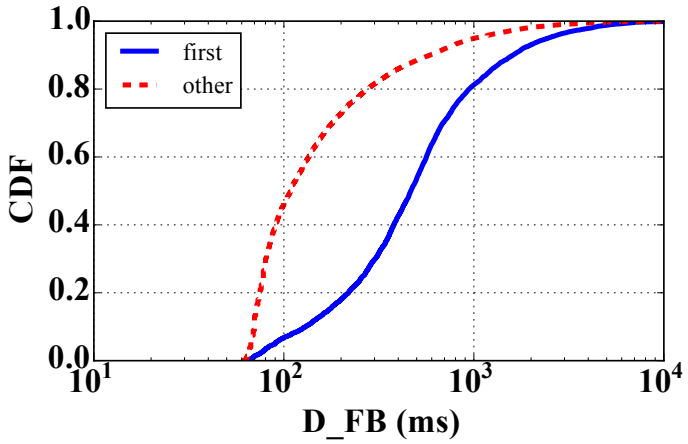


Figure 3.18: D_{FB} (ms) of first vs. other chunks in equivalent performance conditions.

3.4.4 Client's Rendering Stack

1. Avoiding dropped frames requires at least $1.5\frac{sec}{sec}$ download rate. In a typical video session, video chunks include multiplexed and encoded audio and video. They need to be de-multiplexed, decoded, and rendered on the client's machine, which takes processing time. Figure 3.19 shows the fraction of dropped frames versus average download rate of chunks. We define the average download rate of a chunk as video length (in seconds) over total download time ($\frac{\tau}{D_{FB}+D_{LB}}$). A download rate of $1\frac{sec}{sec}$ is barely enough: after receiving the frames, more processing is needed to decode frames for rendering. Increasing the download rate to $1.5\frac{sec}{sec}$ enhances the framerate; however, increasing the rate beyond this does not improve the framerate.

To see if this observation can explain the rendering quality, we look at the framerate as a function of chunk download rate: 85.5% of chunks have low framerate ($> 30\%$ drop) when the download rate is below $1.5\frac{sec}{sec}$ and good framerate when download rate is at least $1.5\frac{sec}{sec}$. About 5.7% of chunks have low rates but good rendering, which can be explained by the buffered video frames that hide the effect of low rates. Finally, 6.9% of chunks have low framerate despite a minimum download rate of $1.5\frac{sec}{sec}$, not confirming the hypothesis. However, this could be explained as follows: First, the av-

erage download rate does not reflect instantaneous throughput. In particular, earlier chunks are more sensitive to changes in throughput, since fewer frames are buffered at the player. Second, when the CPU on the client machine is overloaded, software rendering can be inefficient irrespective of the chunk arrival rate.

Figure 3.20 shows a simple controlled experiment, where a player is running in the Firefox browser on OS X with eight CPU cores, connected to the server using a 1Gpbs Ethernet link. The first bar represents the per-chunk dropped rate while using GPU decoding and rendering. Next, we turned off hardware rendering; we see increase in frame drop rate with background processes using CPU cores.

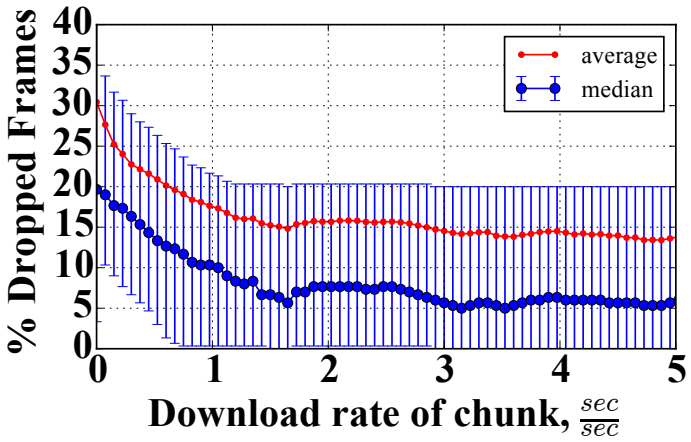


Figure 3.19: %Dropped frames vs. chunk download rate, first bar represents hardware rendering.

2. Higher bitrates have better rendered framerate. Higher bitrates contain more data per frame, thus imposing a higher load on the CPU for decoding and rendering in time. We expect chunks with higher bitrates to have more dropped frames as a result. We did not observe this in our data. However, we observed the following trends in the data: (1) higher bitrates are often requested in connections with lower RTT variation: SRTTVAR across sessions with bitrates higher than 1Mbps is $5ms$ lower than the rest. Less variation may result in fewer frames delivered late. (2) higher bitrates are often requested in connections with lower retransmission rate:

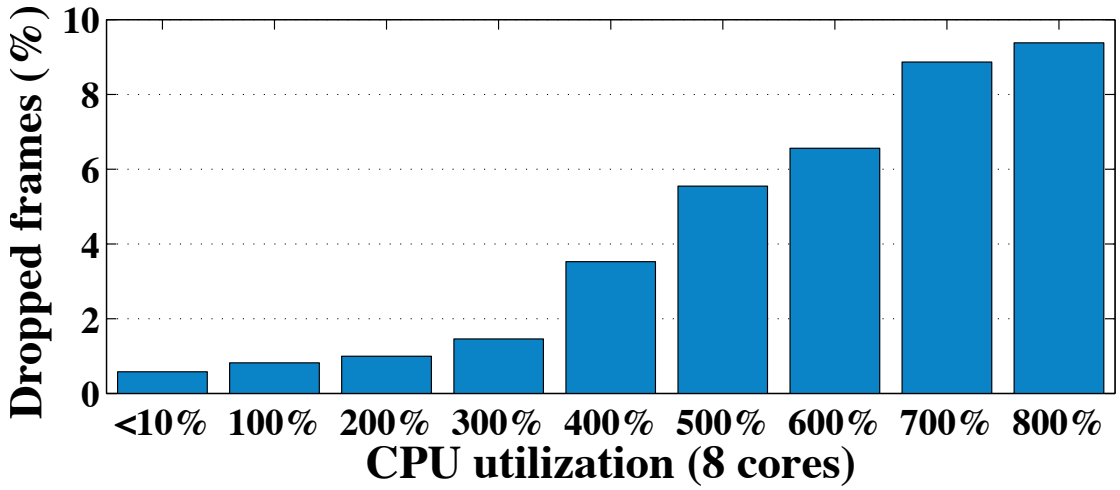


Figure 3.20: Dropped frames per CPU load in a controlled experiment.

the retransmission rate among sessions with bitrates higher than 1Mbps is 1% lower than the rest. Lower packet loss rate results in less frames dropped or arrived late.

3. Less popular browsers have low rendering quality. If we limit our analysis to chunks with good performance ($rate > 1.5\frac{sec}{sec}$) where the player is visible (i.e., $vis = true$), the rendering quality can still be bad due to inefficiencies in client's rendering path. Since we cannot measure the client host environment in production, we only characterize the clients based on their OS and browser.

Figure 3.21 shows the fraction of chunks requested from browsers on OS X and Windows platforms (each platform is normalized to 100%), as well as the average fraction of dropped frames among chunks served by that browser. Browsers with internal Flash (e.g., Chrome) and native HLS support (Safari on OS X) outperform other browsers (some of which may run Flash as a process, e.g., Firefox's protected mode). Also, the unpopular browsers (grouped as Other) have the lowest performance. We further break them down as shown in Figure 3.22. We restrict to browsers that have processed at least 500 chunks. Yandex, Vivaldi, Opera or Safari on Windows have low rendered framerate compared to other browsers.

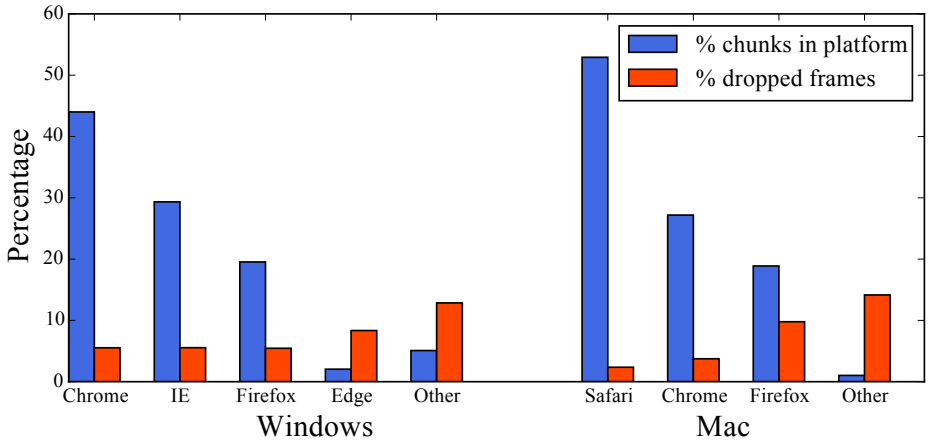


Figure 3.21: Browser popularity and rendering quality in the two major platforms: Windows vs Mac.

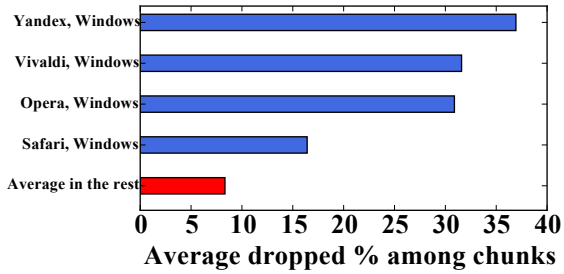


Figure 3.22: Dropped % of (browser, OS), $rate \geq 1.5\frac{sec}{sec}$, $vis = True$.

Take-aways: De-multiplexing, decoding and rendering video chunks could be resource-heavy on the client machine. In absence of hardware (GPU) rendering, the burden falls on CPU to process frames efficiently; however, the resource demands from other applications on the host can affect the rendering quality. We found that video rendering requires processing time, and that a video arrival rate of $1.5\frac{sec}{sec}$ could be used as a rule-of-thumb for achieving good rendering quality. Similar to download stack problems, rendering quality differs based on OS and browser. In particular, we found unpopular browsers to have lower rendering quality.

3.5 Discussion

Monitoring and diagnosis is a challenging problem for large-scale content providers due to insufficient instrumentation or measurement overhead limitations. In particular, (1) sub-chunk events such as bursty losses will not be captured in per-chunk measurements; capturing them will impact player's performance, (2) SRTT does not reflect the value of round-trip time at the time of measurement, rather is a smoothed average; vanilla Linux kernels only export SRTTs to userspace today. To work with this limitation, we use methods discussed in Section 3.4.2, (3) the characterization of the rendering path could improve by capturing the underlying resource utilization and environment (e.g., CPU load, existence of GPU), and (4) in-network measurements help further localization. For example, further characterization of network problems (e.g., is bandwidth limited at the core or the edge?) would have been possible using active probes (e.g., traceroute or ping) or in-network measurements from ISPs (e.g., link utilization). Some of these measurements may not be feasible to collect at Web-scale, because collecting fine-grained TCP metrics can quickly become costly and impose scalability issues. We will tackle these challenges in the next chapter.

3.6 Related Work

Video streaming characterization: There is a rich area of related work in characterizing video-streaming quality. [78] uses ISP packet traces to characterize video while [112] uses CDN-side data to study content and Live vs VoD access patterns. Client-side data and a clustering approach is used in [58] to find critical problems related to user's ISP, CDN, or content provider. Popularity in user-generated content video system has been characterized in [24]. Our work differs from previous work by collecting and joining fine-grained per-chunk measurements from both sides and di-

rect instrumentation of the video delivery path, including the client's download stack and rendering path.

QoE models: Studies such as [37] have shown correlations between video quality metrics and user engagement. [63] shows the impact of video quality on user behavior using quasi experiments. Network data from commercial IPTV is used in [93] to learn performance indicators for users QoE, where [2] uses in-network measurements to estimate QoE for mobile users. We have used the prior work done on QoE models to extract QoE metrics that matter more to clients (e.g., the re-buffering and startup delay) to study the impact of performance problems on them.

ABR algorithms: The bitrate adaptation algorithms have been studied well, [40] studies the interactions between HTTP and TCP, while [3] compares different algorithms in sustainability and adaptation. Different algorithms have been suggested to optimize video quality, in particular [59, 98] offer rate-based adaptation algorithms, where [55] suggests a buffer-based approach, and [113] aims to optimize quality using a hybrid model. Our work is complementary to these works, because while an optimized ABR is necessary for good streaming quality, we showed problems where a good ABR algorithm is not enough and corrective actions from the content provider are needed.

Optimizing video quality by CDN selection: Previous work suggests different methods for CDN selection to optimize video quality, for example [101] studies policies and methods used for server selection in Youtube, while [62] studies causes of inflated latency for better CDN placement. Some studies [65, 46, 44] make the case for centralized video control planes to dynamically optimize the video delivery based on a global view while [8] makes the case for federated and P2P CDNs based on content, regional, and temporal shift in user behavior.

3.7 Conclusion

In this section, we presented the first Web-scale end-to-end measurement study of Internet video streaming to characterize problems located at a large content provider's CDN, Internet, and the client's download and rendering paths. Instrumenting the end-to-end path gives us a unique opportunity to look at multiple components together during a session, at per-chunk granularity, and to discover transient and persistent problems that affect the video streaming experience. We characterize several important characteristics of video streaming services, including causes for persistent problems at CDN servers such as unpopularity, sources of persistent high network latency, and persistent rendering problems caused by browsers. We draw insights into the client's download stack latency (possible at scale only via end-to-end instrumentation); and we showed that the download stack can impact the QoE and feed incorrect information into the ABR algorithm. We discussed the implications of our findings for content providers (e.g., pre-fetching subsequent chunks), ISPs (establishing better peering points), and the ABR logic (e.g., using apriori observations about client prefixes). One of the major limitations that Diva faces is working with aggregated network statistics collected from the kernel of CDN servers. We will address this challenge in the next section.

Chapter 4

Data-plane Performance Diagnosis of TCP

4.1 Introduction

One of major limitations of Diva is that collecting fine-grained TCP logs at end-hosts (i.e., the CDN server's kernel) is costly. Patching kernel to collect TCP metrics (e.g., Web10G [105]) for monitoring consumes the resources and slows down the servers. Frequently snapshotting TCP metrics from the kernel (e.g., `tcp_info`) needs a lot of storage. Therefore, in Diva we could only collect TCP metrics at a low frequency. Also, the collected information is often aggregated and insufficient; hence it lacks flexibility for performance diagnosis (e.g., SRTT instead of individual RTT samples).

In contrast, collecting TCP metrics on the switches in the core of the network does not impose an overhead to the end-hosts. But switches in the core of the network do not have visibility into the end-to-end metrics, and are not owned by the CDN.

Instead, we believe measurement at the "edge"—in the NIC, or top-of-rack switch, as shown in Figure 4.1—offers the best viable alternative. The edge device (i) sees all of a TCP connection's packets in both directions, (ii) can closely observe the

application's interactions with the network without cooperation from the servers, and (iii) can measure end-to-end metrics from the end-host perspective (e.g., path loss rate) because it is only one hop away from the end-host.

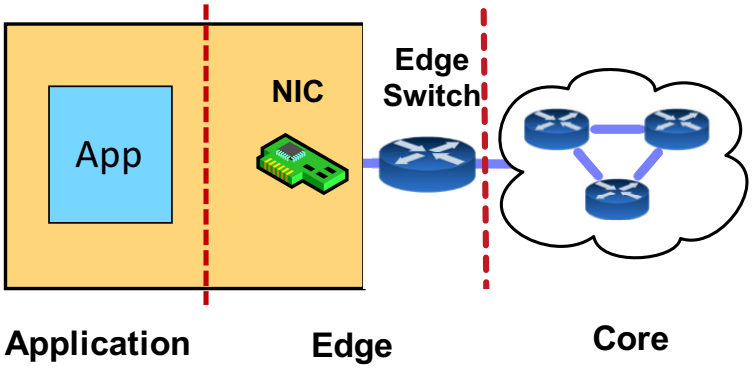


Figure 4.1: Dapper monitors performance at the edge of the network.

Fortunately, emerging edge devices offer flexible packet processing at line rate, in software switches [88], NICs [100], and hardware switches [15]. There have been a variety of proposals to achieve stateful programmable dataplanes, including POF [94], OpenState [12], Domino [92] and P4 [14]. New capabilities, such as flexible parsing and registers that maintain state open up the possibility of detecting and diagnosing TCP performance problems directly in the data plane. Still, a major challenge for data-plane connection diagnosis is finding a "sweet spot" that balances the need for *fine-grained* metrics for diagnosis, while remaining *lightweight* enough to run across a wide range of devices with limited capabilities.

In this section we present Dapper, a **Da**ta-**P**lane **Per**formance diagnosis tool that infers TCP bottlenecks by analyzing packets in real time at the network edge, as shown in Figure 4.2. Dapper quantifies the contribution of the sender, network, and receiver to poor end-to-end performance. Table 4.1 shows examples of problems that can limit a TCP connection's performance. Identifying the entity responsible for poor performance is often the most time-consuming and expensive part of failure detection and can take from an hour to days in data centers [4]. Once the bottleneck is correctly

identified, specialized tools within that component can pinpoint the root cause. To achieve this goal, we need to infer essential TCP metrics. Some of them are easy to infer (e.g., counting number of bytes or packets sent or received), while others are more challenging (e.g., congestion and receive windows).

Dapper analyzes header fields, packet sizes, timing, and the relative spacing of data and ACK packets, to infer the TCP state and the congestion and receive window sizes. A unique challenge that Dapper faces is that the end-hosts may use different versions of TCP, each possibly with tuned parameters. Thus, our techniques must be applicable to a heterogeneous set of TCP connections. The advantage of a data-plane performance diagnosis, apart from line-rate diagnosis, is that the data plane can use this information for quick decision making (e.g., load balancing for network-limited connections). However, a data-plane monitoring tool often has more resource constraints: limited state and limited number of arithmetic and Boolean operations per packet. We discuss design challenges and the necessary steps for Dapper to run in the data plane, including reducing the accuracy of some measurements (e.g., RTT) to lower the amount of per-flow state, as well as two-phase monitoring where we switch from collecting *lightweight* metrics for *all* flows to only collecting *heavyweight* metrics for *troubled* ones.

Roadmap: Section 4.2 explains the TCP performance bottlenecks we identify and how to infer the metrics necessary to detect them. Section 4.3 explains how Dapper diagnoses performance problems from the inferred statistics. Section 4.4 explains how to monitor TCP connections in real time using commodity packet processors programmed using P4 [14, 76]. Section 4.5 discusses our two-phase monitoring to reduce the memory overhead in the data plane. Section 4.6 evaluates the overhead and performance of our system. Section 4.7 discusses the limitations of our approach and implementation. Section 4.8 discusses related work, and Section 4.9 concludes this section.

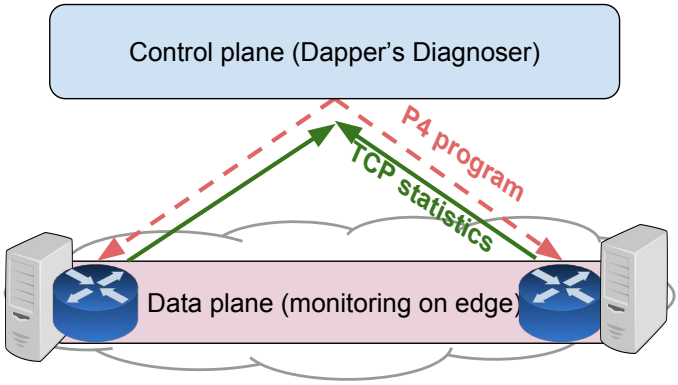


Figure 4.2: Dapper's architecture : (1) data plane monitoring on edge, (2) control plane diagnosis techniques.

4.2 TCP Performance Monitoring

A TCP connection may have bottlenecks at the sender, receiver, or the network. In Table 4.1, we present several examples of performance problems that may arise at each location. With many applications and TCP variants, it is challenging to decide what minimal set of metrics to collect that are both affordable (i.e., does not consume a lot of resources) and meaningful (i.e., helps in diagnosis). In this section, we discuss the metrics we collect to diagnose the performance bottlenecks at each component, and the streaming algorithms we use to infer them.

Location	**Performance Problems**
Sender	slow data generation rate due to resource constraints, not enough data to send (non-backlogged)
Network	congestion (high loss and latency), routing changes, limited bandwidth
Receiver	delayed ACK, small receive buffer

Table 4.1: TCP performance problems at each component

We denote a TCP connection by a bi-directional 4-tuple of server and client IP addresses and port numbers. We focus on the performance of the data transmission from the server, where the server sends data and receives ACKs, as shown in Fig-

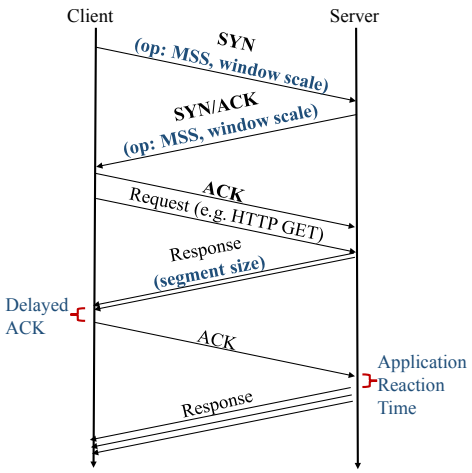


Figure 4.3: Tracking options, segment size, and application reaction time for a simplex connection (server-client)

ure 4.3. Hence, we monitor traffic close to the server, for several reasons. First, being close to the server enables us to imitate the internal state of the server's congestion control and monitor how quickly the server responds to ACKs. Second, since the connection is end-user facing, we do not have access to the client's edge in a real-world deployment. Finally, monitoring at one end reduces the overhead and avoids redundancy as opposed to keeping per-flow state at both ends. Note that if we could instrument and monitor both ends of a connection, it would offer higher accuracy and better visibility into the connection's state by collecting more statistics on the client-side (e.g., delayed ACKs).

4.2.1 Inferring Sender Statistics

Performance problems at the sender limit the TCP connection's performance; for example, an application that is constrained by its host's resources (e.g., slow disk or limited CPU) may generate data at a lower rate than the network or receiver accept, or it simply may not have more data to send; this application is referred to as *non-backlogged*. To find such problems, we measure the *ground-truth* by counting sent

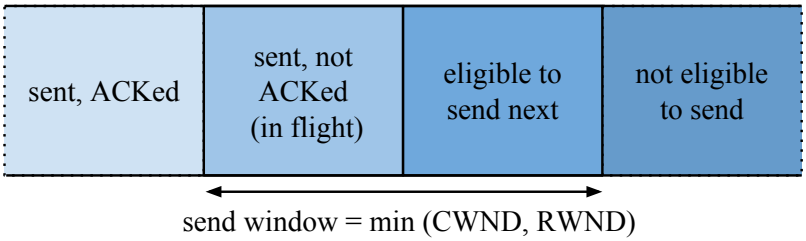


Figure 4.4: Tracking a TCP connection's flight size

packets and compare it with the connection's *potential* sending rate, determined by the receive and congestion windows.

If we could directly monitor the TCP send buffer inside the server's end-host, we could easily observe how much data the application writes to the buffer, how quickly the buffer fills, and the maximum buffer size. In the absence of OS and application logs from the servers, we rely on two independent metrics to see if the application is sending "too little" or "too late": (i) *generated segment sizes*, to measure if an application is sending too little, and (ii) *application's reaction time*, to see if it is taking too long.

Inferring flight size to measure sending rate: Upon transmission of new packets, we update the packet and byte counters for each connection and measure the connection's flight size to infer its sending rate (i.e., $\frac{flight\ size}{RTT}$). The flight size of a connection is the number of outstanding packets—packets sent but not ACKed yet—as shown in Figure 4.4, and is inferred via examining the sequence number of outgoing packets and the incoming acknowledgment numbers to track how many segments are still in flight. In this figure, the "send window" represents the available window to the sender, that is, the maximum packets that the sender can send before waiting for a new ACK; this window is limited by both the RWND and CWND. In this example, the application is not backlogged, because it has not fully used the available window. We can see that flight size can at most be equal to the send window.

Note that the flight size of a connection is an important metric in diagnosis because it depends on the congestion window, the receive window, and the application's own data-generation rate (i.e., is the application backlogged?). We will revisit this metric in Section 4.2.2.

Extracting MSS from TCP options: The segment sizes in a connection indicate the amount of data that the sending application generates. We infer MSS by parsing the TCP options exchanged during the three-way handshake, as shown in Figure 4.3, in the SYN and SYN-ACK packets.

Measuring sender's reaction time via cross-packet analysis: We define the sender's reaction time as the time window between the arrival of a new acknowledgment and the transmission of a new segment. The reaction time evaluates the sending application's data-generation rate, i.e., whether or not it is backlogged. Lower reaction times indicate that the data was already processed and was just awaiting an opportunity to be sent. We measure the reaction time using time-stamps of incoming acknowledgments and outgoing packets. We compare the reaction time with an empirically derived threshold, calculated based on the latency between the edge and the server.

Note that "cross-packet analysis" happens at the edge, thus the reaction time consists of the application's own data-generation latency plus the communication latency between the application and edge (i.e., latency in the hypervisor and virtual switch). More importantly, the network latency does not influence this metric. Also, notice that the edge acts as *the single point of observation*, seeing both directions of the flow to do cross-packet analysis. These two conditions are not necessarily true on the switches in the core of network to measure the sender's reaction time since the packets could have been delayed at earlier network hops—as opposed to the application itself—or worse yet, taken different paths.

4.2.2 Inferring Network Statistics

Network problems cause poor performance in a TCP connection. For example, in a congested network, the increased packet loss and path latency cause the sender's TCP congestion-control algorithm to decrease the sending rate. The sender's reaction to congestion varies based on the congestion control in use (e.g., Reno vs Cubic) and the severity of the congestion itself. To determine and quantify the network limitation in the performance of a TCP connection, we need to measure and compare its congestion window—how much the network allows the connection to send—against the receive window—how much the receiver allows the connection to send—and how much data the sender has available to send.

To infer a connection's congestion window, we "imitate" the internal congestion-control algorithm of the server by tracking flight size, packet losses and their kinds, and duplicate ACKs. We track RTT and RTTvar to pinpoint the effect of network congestion, routing changes, and queuing delay on TCP performance.

Inferring loss via retransmission: We calculate a connection's loss rate by counting the lost and total packets in the flow. In addition, we use a counter to track the duplicate ACKs and use it to infer the kind of loss: fast-retransmission (FR) is triggered after a fixed number of duplicate ACKs are received (normally 3); however, a timeout is triggered when no packet has arrived for a while (RTO). We track the sequence number of outgoing packets (i.e., packets sent by the server) which helps us find the retransmission of previously seen sequence numbers. Using the duplicate ACK counter, if we see at least three duplicate acknowledgments before a re-transmission, we treat it as a fast retransmission, otherwise, we deduce that the loss was recovered by a timeout.

Estimating latency by passive RTT measurements: We use Karn's algorithm [60] to estimate SRTT, SRTTvar, and RTO using a series of "RTT measure-

ments". An RTT measurement is the time between when a segment was sent and when its acknowledgment reached the sender. For any connection, upon transmission of a new segment, we create a (time-stamp, sequence number) tuple and maintain it in a queue. Upon arrival of an ACK, we inspect the queue to see if any of the tuples are acknowledged by it. If so, we create an RTT measurement and use it to update the latency statistics via Karn's algorithm.

Note that a retransmitted packet cannot be used as an RTT measurement, because the corresponding ACK cannot be correctly mapped to a single outgoing time-stamp. Also, if a connection has multiple outstanding packets, the queue will have multiple tuples, i.e., the length of the queue grows with the flight size. Finally, if an incoming ACK acknowledges multiple tuples (i.e., a delayed ACK) we de-queue multiple tuples but only create one RTT measurement, from the most recent tuple, to exclude the effect of delayed ACK on RTT.

Estimating congestion window via flight size and loss: It is challenging to estimate a connection's congestion window outside the server's networking stack due to following reasons: *(1) Many TCP variations:* There are many different TCP congestion control algorithms used today (e.g., Reno, New Reno, Cubic, DCTCP). Some are combined with tuning algorithms (e.g., Cubic combined with HyStart to tune `ssthresh`) or have configurable parameters (e.g., initial window). *(2) Thresholds change:* `ssthresh` is the threshold that separates slow-start (SS) from congestion-avoidance (CA) in the TCP state machine and is initially set to a predefined value. However, the Linux kernel *caches* the `ssthresh` value to use it for similar connections in the future. Also, `ssthresh` changes throughout the life of a connection (e.g., under packet loss). Therefore, if the full history of a connection (and even *past* connections!) is not observed, `ssthresh` is unknown, making it impossible to detect transitions from SS to CA based on the `ssthresh` threshold.

In the presence of these challenges, we rely on these TCP invariants to infer congestion window: *(1)* The *flight size* of a connection is bounded by congestion window, as shown in Figure 4.4. We denote this lower-bound estimate of congestion window with inf_cwnd. In a loss-free network, inf_cwnd is a moving maximum of flight size of the connection. Note that in the absence of loss, if the connection's flight size decreases, it is either due to the sender producing less data (i.e., not fully utilizing the window) or the receiver's limited receive window; hence, inf_cwnd does not decrease. *(2)* If a packet is *lost*, we adjust inf_cwnd based on the nature of loss, a timeout resets it to IW and a fast-retransmit causes a multiplicative decrease.

Estimating the congestion window based on these invariants makes inf_cwnd"self-adjustable", working regardless of TCP variant and configuration, and is calculated according to Algorithm 1. Algorithm 1 consists of a while loop that inspects every new packet. If a new segment is transmitted and the connection's flight size grows beyond inf_cwnd, we update inf_cwnd to hold the new maximum value of flight size. In case of retransmissions (loss), we decrease inf_cwnd by the multiplicative decrease constant, C, if loss is recovered by fast recovery. Otherwise, inf_cwnd is reset to initial window if recovered by a timeout. In this algorithm, we assume the CDN knows the value of C and IW, or they can be easily inferred, either indirectly via observing how large the first window is and how it changes after a loss, or directly via tools such as Nmap [74].

Note that inf_cwnd as estimated by Algorithm 1 does not require full knowledge of connection's history, thresholds, or the congestion control algorithm, and is only dependent upon measuring the connection's flight size and loss, thus solves the challenges above without cooperation of servers or using the end-host's resources[1].

[1]To keep our heuristics general across all TCP variants, we do not rely on selective acknowledgments.

Algorithm 1: Estimating inf_cwnd

Input: multiplicative decrease factor (C), initial window (IW)
Output: inf_cwnd

```
1  while P ⟵ capture new packet do
2      if P is new segment and flight size ¿ inf_cwnd then
3          inf_cwnd⟵ flight size
4      else if P is retransmitted then
5          if fast retransmit then
6              if first loss in fast recovery then
7                  inf_cwnd⟵ C×inf_cwnd
8          else if timeout then
9              inf_cwnd⟵ IW
```

4.2.3 Inferring Receiver Statistics

The receiver-side of a TCP connection can limit the flow by decreasing its advertised window (i.e., RWND) or slowing down the rate of acknowledgments [115] to control the release of new segments.

Tracking RWND per-packet and per-connection : To quantify the receiver limitation in a TCP connection, we track the advertised RWND value *per-packet*, reflecting how much buffer is available on the client. We also track the *per-connection* agreed upon window scaling option during the TCP handshake, as shown in Figure 4.3, which is used for scaling RWND.

Inferring delayed ACKs via RTT samples: When an incoming ACK acknowledges multiple tuples in the queue, it must be a delayed ACK, as the client is acknowledging multiple segments at once. When we de-queue tuple(s) based on an incoming ACK, we count and average the number of de-queued tuples per ACK to reflect the effect of delayed ACK.

Summary: Figure 4.5 summarizes how Dapper updates a TCP connection's performance statistics while processing a new packet. The packets are first hashed on the

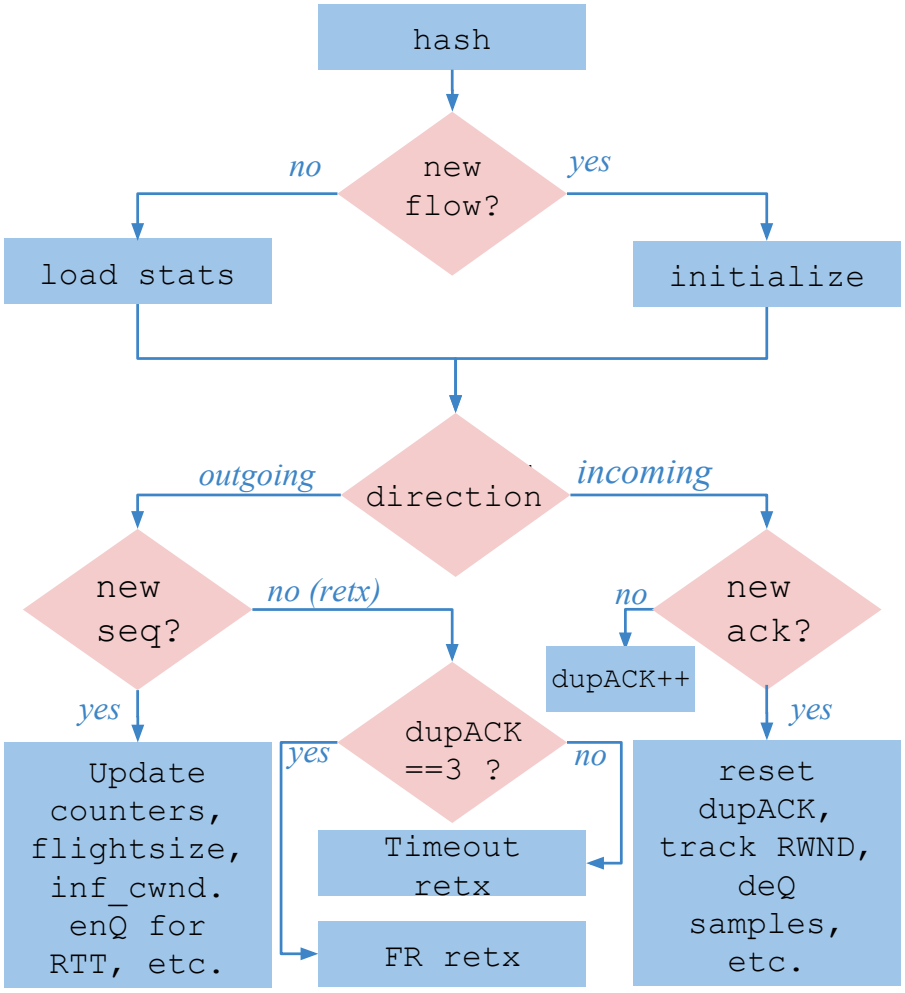


Figure 4.5: Dapper's packet-processing logic

four-tuple to either initialize a new flow, or read the existing statistics. Then, based on the direction of the packet, the relevant header fields are extracted and used to update the metrics. The blue boxes show the analytics performed to keep per-flow state (e.g., update flight size) and the pink diamonds show the conditions used to decide which state to update.

4.3 TCP Diagnosis Techniques

In this section, we describe how Dapper uses the statistics gathered by the streaming algorithms discussed in section 4.2. The high-level objective of the diagnosis techniques is to troubleshoot a connection's performance limitation.

Diagnosing Sender Problems: Our goal is to find if the sender-side is not limiting the connection's performance (backlogged), or limiting the sending rate via not having enough data to send or taking too long to produce it (non-backlogged).

1. Exponential sending rate indicates a backlogged sender: On transmission of new segments, we first examine the connection's macroscopic behavior, i.e., the sending rate, and check to see if it is growing exponentially to infer if the connection is in slow start. More accurately, we compare the relationship between ACKs and the data packets to see how many packets the sender transmits after a new ACK. If sending rate grows exponentially, we know the connection is *not sender-limited*. Otherwise, we attempt to understand if the connection is sender-limited by checking the next heuristics.

2. If the sender is backlogged, it will "completely" use the send window: When sending rate does not grow exponentially, the connection could either be in congestion avoidance with a backlogged sender, or it could suffer from a non-backlogged sender, producing less data than CWND. This heuristic checks to see if the connection's flight size is consistently less than the allowed window to send, determined by the minimum of RWND and CWND, i.e., if $flightsize < min(inf\_cwnd, RWND)$. If so, the connection's performance is limited because the sending application does not send more, not because it's not allowed to.

3. Sending less than allowed, or later than allowed, indicates a non-backlogged sender: Here we examine the connection's microscopic behavior to see if the connection is under-utilizing the network, not concerning the "number" of packets in flight like the

previous heuristic, but instead focusing on the "size" and "timing" of packets. More concretely, we check to see if a connection is sending packets that are smaller than MSS, or if the application's reaction time (i.e., data generation time) is larger than an empirically derived threshold for backlogged applications. If either of these conditions are met, we conclude that the application is non-backlogged, hence the connection is *sender-limited* indicating that the sender is either not generating enough data to fill up a whole packet, or not responding almost immediately when it is allowed to send.

4. How the flight size changes during a loss recovery gives us a clue of how backlogged the sender is: In addition to the heuristics above, during loss recovery a connection reveals some information about its internal state[2]. As a reminder, fast-recovery causes the congestion window to decrease by a multiplicative factor, C.

Consider a connection not limited by receiver, and assume that the sending application's data generation rate remains unchanged during the network loss. We denote the flight size of the connection before loss by f_1 and after the loss by f_2. Figure 4.6 shows three example scenarios, where loss happens at 30ms, prompting CNWD to decrease by half. The $\frac{f_2}{f_1}$ ratio gives us the following insights: if flight size is closely tracking CWND, $\frac{f_2}{f_1} = C$, the sender is backlogged (app 1); if the connection was not fully using the CWND before loss but is backlogged after the loss, $C < \frac{f_2}{f_1} < 1$ (app 2); finally, if the flight size remains unchanged the sender is not backlogged (app 3). Note that in these examples we assume the state of the sender remains unchanged during the loss recovery.

Diagnosing Network Problems: Our goal is to determine if the network is restricting TCP performance, either due to limited bandwidth (congestion window limited), high packet loss rate, or increased latency due to problems such as queuing delay or routing problems.

[2]This heuristic can be treated as a bonus, and the diagnosis algorithm does not rely on seeing a loss.

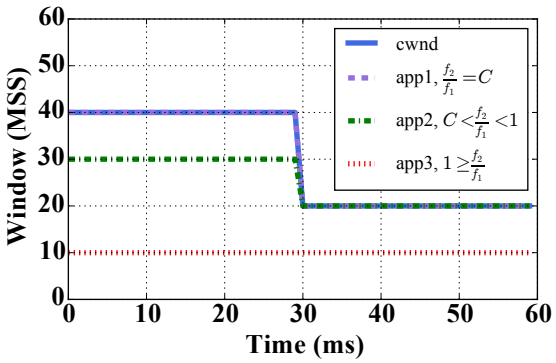


Figure 4.6: Flight size before and after loss

1. Small congestion window hurts TCP's performance: When the network has performance constraints, for example limited network bandwidth, the congestion window will limit the rate of the connection, that is: $\textit{flight size} \leq inf\_cwnd < RWND$. Upon a packet retransmission, if loss causes the inf_cwnd value to drop below the RWND, we deduce the connection is limited by network.

2. Increased network path latency slows TCP's rate: The sending rate of the connection is a function of both the flight size and RTT; the sender can only increase the window after a new ACKs arrives, which usually takes an RTT. To track the impact of network latency on TCP performance, we use the RTT measurements as explained in section 4.2. The cloud provider can either define an "expected RTT" per TCP connection based on SLAs, or use the minimum RTT sample per-flow as the baseline. To diagnose path latency problems, we compare the RTT values with the expected RTT to detect if a connection's latency is acceptable.

Diagnosing Receiver Problems: Our goal is to find if the receiver is restricting TCP performance, either by offering a small receive buffer (receive window), or by delaying ACKs.

1. Small receive window hurts performance: Upon updating the RWND and inf_cwnd values, this heuristic compares them with the current flight size to see if the connec-

tion's sending rate is receiver-limited, that is: *flight size* $\leq RWND < inf\_cwnd$. If so, the connection is diagnosed as receiver-limited.

2. Delayed acknowledgment hurts performance: The receiver can limit TCP performance by sending ACKs with a delay; delayed ACKs has been shown to cause issues in datacenters [115]. When the client sends ACKs with a delay, for example, sending acknowledgments for every other packet, the sender's opportunity to increase its window is halved[3]. For each connection, we measure the average number of RTT samples freed by each new ACK and if the average is greater than one, we diagnose the connection as receiver-limited due to delayed acknowledgment.

4.4 Data-Plane Monitoring

In this section, we describe how Dapper tracks TCP connections in the data plane and discuss the principles behind our target-independent solution using P4. We outline the P4 features (e.g., metadata) that enable us to monitor TCP connections according to Figure 4.5 (Section 4.4.1); then, we discuss the target-specific resource constraints and how to mitigate them (Section 4.4.2).

4.4.1 TCP Monitoring Prototype in P4

P4 is a programming language that allows us to express how packets are processed and forwarded in a target-independent program, therefore, our P4 prototype can run in a public cloud on a variety of targets, as long as at least one of the elements at the edge (the switch, the hypervisor, or the NIC) can run P4 programs.

To monitor TCP connections in the data plane in real time, we need to extract and retain packet header information (P4's flexible parsing), carry information across multiple stages of packet processing (P4's metdata), and store state across successive

[3]Note that the congestion window on sender-side opens up upon receiving each new ACK, as every ACK is a sign that a packet has left the network, hence the network can receive more.

packets of the flow (P4's registers). Furthermore, to realize the logic in Figure 4.5, we need to perform specific operations on each packet, shown with blue boxes (P4's tables and actions) and check test conditions based on both the packet headers and the flow state to invoke the relevant tables, shown by pink diamonds (P4's flow control).

1. Extracting headers and options via flexible parsing: Using header definitions, we identify the relevant header fields in a packet. In our prototype we assume the TCP packets have Ethernet, IPv4, and TCP headers, although this can be easily extended to include other protocols (e.g., IPv6). The following snippet shows some relevant TCP headers. In addition, we provide a parser that extracts headers (e.g., source and destination IP from the IP header).

```
header_type tcp_t {
    fields {
        srcPort : 16;
        dstPort : 16;
        seqNo : 32;
        ackNo : 32;
        ...
        }
}
```

P4 models the parser as a state machine represented by a parse graph. The parsed headers need to be "de-parsed", i.e., merged back, to a serial stream of bytes before forwarding. TCP options require TLV (Type-Length-Value) parsing. For parsing options, we use "masks" to identify the "type" (e.g., type 2 represent MSS), then a parser is called to extract that option knowing its "length" (e.g., `parse_mss` for MSS in the snippet below) which returns the control back to the original parser when done. This creates a loop in the parsing graph causing the exact de-parsing behavior to be undefined. To solve it, we impose a fixed order for de-parsing, by using the *pragma* keyword as the following code snippet shows. We will use this extracted MSS value in the subsequent parts.

```
@pragma header_ordering ethernet ipv4 tcp options_mss options_sack options_ts options_nop
    options_wscale options_end

parser parse_tcp_options {
   return select(mymeta.opt_counter, current(0,8)) {
      ...
      0x0002 mask 0x00ff : parse_mss;
        ...
   }
}
```

2. Keeping per-flow state in registers: Registers are stateful memories, which are essential to Dapper because they maintain the per-flow state as it gets updated after processing each packet. Registers consume resources on the target, hence are a major limitation in running our solution on specific targets, therefore we will minimize the required per-flow state to ensure our program runs on commodity hardware in Section 4.5.

P4 registers can be global, referenced by any table, or static, bound to a single table. The following code shows one of our global registers, `MSS`, as an array of 16-bit values, and `instance_count` is the number of entries in the flow table. Each packet is hashed to find its flow index in the register array. We will explain our bi-directional flow hashing in more details shortly.

```
register MSS {
    width : 16;
    instance_count : ENTRIES;
}
```

When tracking a TCP connection for diagnosis, some register values depend on the value of other registers; for example, only by comparing a packet's acknowledgment with previous ACKs of the flow can we detect a duplicate ACK. To update such dependent registers, we have to *read* other register(s), test *conditions*, and finally *update* the target register.

3. Carrying information per-packet via metadata: Metadata is the state associated with each packet, not necessarily derived from the packet headers, and can be used as temporary variables in the program. We use metadata to carry the information belonging to the same packet from one table to the other. The code below shows the most widely used metadata in our program, `flow_map_index`, which is the flow's index produced by hashing. This metadata carries the index over to the subsequent tables, each using it to index their registers for read/write. Below, we show a code snippet for declaring metadata field named `flow_map_index` where `FLOW_MAP_SIZE` indicates its width in bits. In the next subsection, we explain how we use this metadata in hashing.

```
header_type stats_metadata_t {
   fields {
    flow_map_index : FLOW_MAP_SIZE; // flow's map index
    ...
   }
}
metadata stats_metadata_t stats_metadata;
```

Some metadata has special significance to the operation of the target (i.e., the standard intrinsic metadata). In particular, we use the target's `ingress_global_timestamp` as the arrival time of the packet, which is necessary to infer latency metrics such as the sender's reaction time and SRTT.

4. Bi-directional hashing using metadata and registers: As discussed earlier, our streaming algorithm must see both directions of traffic to capture our cross-packet metrics, e.g., application reaction time. To do this, we need to hash both directions to the same index and process them as one entity. Unfortunately, P4 provides no primitives or methods for hashing both directions to the same index—no symmetric hashes. Although some targets may allow configuring the hash function through run-time APIs, this support may vary across targets [76]. Therefore, we build our own

symmetric hash using P4's default hash algorithm, e.g., `crc32`, by defining two sets of headers to hash on, with one in the reverse order of the other. In other words, one direction is hashed based on (src IP, dst IP, src Port, dst Port) fields, and the reverse direction is hashed on (dst IP, src IP, dst Port, src Port) fields. To keep the direction's hash function consistent, we use a simple and consistent comparison of the two IPs: if $srcIP > dstIP$, we hash the packet header in the former order, otherwise we hash the packet headers in latter order. This guarantees that each side of packet stream gets consistently hashed by one of these hash functions, but results in the same index value per flow.

5. Realizing operations using actions and tables: To realize the blue boxes in the flowchart of Figure 4.5, P4 tables and actions are used. A P4 table defines the fields to match on and the action(s) to take, should the entry match. P4 tables allow us to express different sets of match-action rules to apply on packets; for example, the set of actions for an outgoing packet differs from incoming packets. Furthermore, some tables could be dedicated to monitoring while others are dedicated to forwarding packets (e.g., `ipv4_lpm` and `forward`). A fundamental difference between our monitoring tables from regular forwarding tables in P4 is that our monitoring tables have a single static entry that matches on every packet —hence, have no match field. In contrast, `ipv4_lpm` is a forwarding table that uses longest prefix matching to find the next hop. The following code snippet shows two of Dapper's tables, the `lookup` table, that hashes every packet to find its flow index, and the `init` table that initializes the flow upon observing its first packet (e.g., saves the extracted MSS value in the MSS register array, at the flow's index).

```
table lookup{
   actions {
      lookup_flow_map;
   }
}
table init{
```

```
    actions {
        init_actions;
    }
}
```

```
action lookup_flow_map() {
 modify_field_with_hash_based_offset(stats_metadata.
 flow_map_index, 0, flow_map_hash, FLOW_MAP_SIZE);
}

action init_actions() {
   register_write(MSS, stats_metadata.flow_map_index, options_mss.MSS);
    ...
}
```

Actions in P4 are declared imperatively as functions, inside the tables. Actions can use registers, headers, and metadata to compute values. An example action is `register_write`, which takes a register, and index, and a value as input, and sets the value of the register array at the index accordingly. Actions are shown with red color in our code snippets.

6. Conditions via control-flow: The control flow of a P4 program specifies in what order the tables are to be applied. Inside the control segment, we can "apply" tables and test conditions. The choice of which block to execute may be determined by the actions performed on the packet in earlier stages. The control flow is what enables us to design the pipeline and implement the conditions (pink diamonds) in P4 as the flowchart shows in Figure 4.5.

In the "widely-supported" P4 specification [76], conditional operations are restricted to the control segments of program; that is, we cannot have `if-else` statements inside a table's logic. Fortunately, P4 offers metadata, which can be used as temporary variables in the program. The metadata gives us an opportunity to read the current value of *conditional registers* inside an earlier table—the "loader"—in the pipeline, store their values in the metadata, test the conditions in the control

section, and apply the appropriate set of tables conditionally. Note that we need the "loader" table because of the current restrictions in P4 that allows conditions only in the control segment.

```
control ingress {
  if ( ipv4.protocol == TCP_PROTO) {
    if( ipv4.srcAddr > ipv4.dstAddr ) {
      apply(lookup);
    }else{
      apply(lookup_reverse);
    }
    if ( (tcp.syn == 1) and (tcp.ack == 0) )//first pkt
      apply(init);
    else
      apply(loader);
    if (ipv4.srcAddr == stats_metadata.senderIP){
      if( tcp.seqNo > stats_metadata.seqNo ){
        apply(flow_sent);
        if(stats_metadata.sample_rtt_seq == 0)
          apply(sample_rtt_sent);//"temp" has the new flightsize
        if(stats_metadata.temp > stats_metadata.mincwnd)
          apply(increase_cwnd);
      }else{
        if(stats_metadata.dupack == DUP_ACK_CNT_RETX)
          apply(flow_retx_3dupack);
        else
          apply(flow_retx_timeout);
      }
    }
    else if(ipv4.dstAddr == stats_metadata.senderIP ) {
      if( tcp.ackNo > stats_metadata.ackNo ){
        apply(flow_rcvd);//new ack
        if( tcp.ackNo >= stats_metadata.sample_rtt_seq and stats_metadata.sample_rtt_seq>0){
          if(stats_metadata.rtt_samples ==0)
            apply(first_rtt_sample);
          else
            apply(sample_rtt_rcvd);
        }
      }else
        apply(flow_dupack);//duplicate ack
    }
```

```
  }
   apply(ipv4_lpm);
   apply(forward);
}
```

4.4.2 Hardware Resource Constraints

In this section, we explain our design choices to monitor connections in P4. These choices stem from a variety of restrictions, in particular, the limited resources on hardware switches, missing features in the P4 spec, and the diversity of hardware targets, which would require us to design for the least common denominator among the supported features.

1. Handling hash collisions: We use 32 bits for hashing in our prototype; regardless, collisions are often a concern in hash tables. In our software implementation, we handle collisions in the hash table by "hash-chaining": we store the four tuple key of the connection and create a linked-list of flows in the same index with different keys. However, since memory in hardware is limited, we decide to not store a connection's tuple. Still, if collisions go undetected they may pollute the accuracy of collected statistics. Hence, it is useful to assess if the accuracy of a connection's statistics has been compromised. Therefore, we perform basic checks on the packet's sequence number versus the flow's previously sent sequence numbers and available windows (i.e., does the sequence number fall within the acceptable window?). This comparison requires additional tables or registers per-flow, but can store the result of conditions in a Boolean variable, named "sanity check". The sanity check can be queried from the data plane along with the connection metrics to indicate whether collected statistics are reliable for diagnosis.

2. Keeping one RTT sample at a time: As discussed in Section 4.3, to accurately track the connection's RTT, we maintain a queue of tuples based on the outgoing

packets, where each tuple is (sequence number, time-stamp). The received ACKs are compared to the tuples of the queue to make an RTT measurement. Since the queue of tuples grows with the flow's flight size, it increases the amount of state per-flow in our P4 program. Because the hardware resources on a switch are limited, we limit the number of outstanding RTT tuples in our P4 program to one at a time, per-flow: we only sample an outgoing packet for RTT if the flow's queue is empty.

3. Multiple accesses per register array: Our program accesses some registers from multiple points to use them in test conditions; e.g. duplicate ACK count register is read in the loader table and used for identifying the kind of loss, and upon a new duplicate ACK another table updates it. Currently, conditional operations in P4 are restricted to the control segments. Thus we cannot avoid accessing some registers in multiple tables. Unfortunately, accessing a register from multiple tables limits the processing rate. However, the ternary operator (`?:`) will be supported in the next P4 version [26], allowing us to perform simple conditional assignments, eliminating the need for global registers, and permitting our solution to run at line rate.

4. Relying on control plane to scale RWND: In TCP, the advertised RWND should be shifted by the window scale (as negotiated in handshake) to calculate the actual receive window. However, most P4 targets can only "shift" by a "constant" value [76]. So, we instead record both values and allow the control plane to query both and perform the shift.

5. Foregoing RTTvar: Calculating RTTvar involves capturing the absolute difference of the smoothed moving average RTT (SRTT) and the current RTT sample (RTT). This difference can be captured via an `abs` operator or by introducing a new comparison test, i.e., a new pipeline stage. Unfortunately, the P4 specification does not support the `abs` operator and adding a new stage impacts the processing rate of

our implementation to gain a single metric. By default, Dapper does not include this stage but can be enabled optionally.

4.5 Two-Phase TCP Monitoring

Our goal in this section is to lower the cost of monitoring and diagnosing TCP connections. We present a two-phase monitoring technique to decrease the amount of state required. The first phase monitors all connections continuously but only collects low-overhead metrics, enough to *detect* but not *diagnose* performance problems. When a connection meets the "badness" criterion, heavier-weight monitoring is enabled to diagnose the poor performance.

Phase 1: Lightweight detection: In the first phase, we collect lightweight metrics that suffice to "detect" the existence of performance problems, based on a badness criterion. The statistics used in the first phase must be: 1. *lightweight* to maintain, ensuring that the continuous monitoring of connections in the data plane is cheap, and 2. *general* enough to capture the badness of the flow, regardless of the component limiting the performance. We use the *average rate of the flow* as an indicator of how well it's performing.

To maintain the average rate of flow, we keep three registers: 1. `init_time`, the time-stamp when monitoring began, 2. `bytes_sent`, total bytes sent so far, and 3. `update_time`, the time-stamp when the flow was updated last. The control plane can query these states per-flow and find the connections that look troubled based on a CDN-wide specific threshold. Upon observing low rate, the CDN operators can turn on the heavy-weight monitoring mode to do diagnosis.

Phase 2: Diagnosis of troubled connections: The second phase is "diagnosis", where we collect heavyweight metrics for a troubled connection to shed light on the component that is hindering the flows performance. These metrics include the com-

plete set of TCP statistics (discussed in Section 4.2) and our diagnosis techniques (discussed in section 4.3). This phase consumes more state on the switch, but note that it is only turned on after a problem is detected, hence the switch state consumption overall decreases. The two-phase monitoring can be thought of as a "long and narrow" table (all flows, few metrics), followed by a "short and fat" table (few flows, many metrics).

There are some challenges with inferring TCP metrics midstream: First, the TCP constants, in particular MSS and window scale are exchanged once, during the handshake. Second, the flow counters (e.g., packets sent, ACKed, or in flight) are unknown. This results in errors in the inferred value of flight size, which is also used to estimate inf_cwnd.

Our solution to these challenges is two fold: 1. Parse and keep the TCP options during the handshake for all the flows in the first phase, in case they are needed later. This approach provides high accuracy but requires more data-plane state. 2. Infer them midstream, only when necessary. Of course, inferring constants midstream reduces the memory overhead at the expense of accuracy. To infer MSS from midstream, we need to keep track of the largest segment sizes seen so far. To infer the window scaling option midstream we track the flight size and the unscaled RWND as advertised in received packets. We use the TCP invariant that "the flight size of a connection is limited by the receive window", hence we can estimate the *lower-bound* window scaling factor:

$$\begin{aligned} \textit{flight size} &\leq \textit{RWND} \cdot 2^{\textit{scale}} \\ \lceil \log_2 \frac{\textit{flightsize}}{\textit{RWND}} \rceil &\leq \textit{scale} \end{aligned} \tag{4.1}$$

Finally, to infer inf_cwnd, we rely on flight size, which itself is inferred from tracking sequence numbers of outgoing packets and incoming ACKs midstream. We will evaluate the accuracy of these metrics to show how close to actual values the accuracy of midstream inferred metrics get.

4.6 Evaluation

In this section we evaluate the accuracy of our heuristics, and the overhead of our P4 prototype for hardware switches, and our C prototype for hypervisors (Section 4.6.2). Then we use the software prototype to validate the accuracy of our diagnosis algorithm using synthetic traffic (Section 4.6.3). Next, we showcase Dapper in the wild by analyzing CAIDA packet traces (Section 4.6.4). Finally, we demonstrate the trade-offs in accuracy and overhead in the P4 design.

4.6.1 Accuracy of Heuristics

We begin the evaluation by comparing our heuristics against the ground truth data collected from within the end-hosts (using Linux kernels `tcp_info` structure). In particular, Figure 4.7 demonstrates how well the CWND is inferred for the first 5 seconds of a flow. The connection belongs to a simple server-client application where the server is transmitting a large file over a 1G link, both machines run TCP Cubic. We make three observations based on this figure: (1) Dapper's inferred CWND closely follows Cubic's. This suggests it is possible to accurately infer the congestion window without tenant's cooperation. (2) If monitoring begins at later times (e.g., after 1 sec in (b) or 3 sec in (c)) Dapper infers CWND accurately, and the inferred CWND converges quickly. (3) We show how losses offer extra information for tracking CWND. Note, that while the figure only presents the first 5 second, our observations generalize to whole flow and to other flows in our dataset.

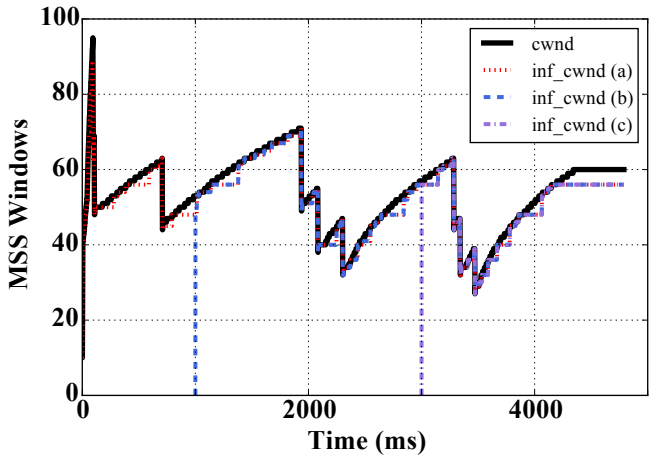


Figure 4.7: First 5 seconds of a flow shows how closely our heuristic tracks CWND, from monitoring starting points of (a) beginning, (b) 1 sec, (c) 3 sec.

4.6.2 CPU and Memory Overhead

We evaluate Dapper's overhead along two dimensions: memory utilization on hardware switches, and CPU Utilization on hypervisor or vswitch; this is mainly because switches operate at line rate but are constrained for memory, while software solutions are not often constrained by memory, but by CPU utilization. Dapper's software prototype is implemented in `C` and uses `libpcap` to capture packets.

Memory in hardware: In single-phase mode, our P4 prototype keeps 67 bytes of state for each connection (i.e., 16 four-byte registers to keep the flow state, a two-byte register to track MSS, and a one-byte register for scale). In addition, 40 bytes of metadata are used to carry a packet's information across tables. In a typical data center, a host can have 10K connections [115], which results in 670 KB of state required to track their state. In the two-phase monitoring prototype, as described in section 4.5, we use the average rate of flows as the badness factor in the first stage, which would impose about 16 bytes per flow in the first stage. Figure 4.8 shows the amount of state needed for single-phase monitoring, as opposed to two-phase monitoring with 10% and 20% troubled connections.

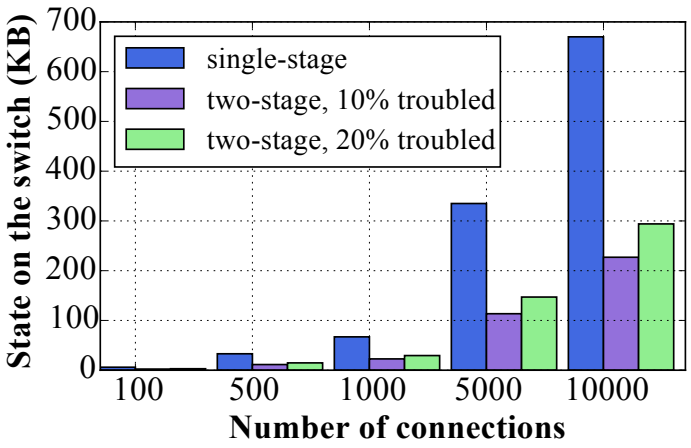


Figure 4.8: Required state on switch in single vs two-phase monitoring, with 10% and 20% troubled connections.

As more connections are monitored, the expected rate of collision in the hash table increases. Figure 4.9 shows the expected collision rate per number of flows (k), for varying sizes of tables (N). Assuming the N hash values are equally possible, the probability that a flow A shares the same index with flow B is $\frac{1}{N}$. So, the probability that the other k-1 flows will not share the same index is $(1 - \frac{1}{N})^{k-1}$, resulting in the expected likelihood of collisions of $1 - (1 - \frac{1}{N})^{k-1}$. Thus, to track 10K connections with a collision rate of less than 4%, we need the table size to be at least $262,144$ (2^{18}), which results in less than 18 MB space on the switch.

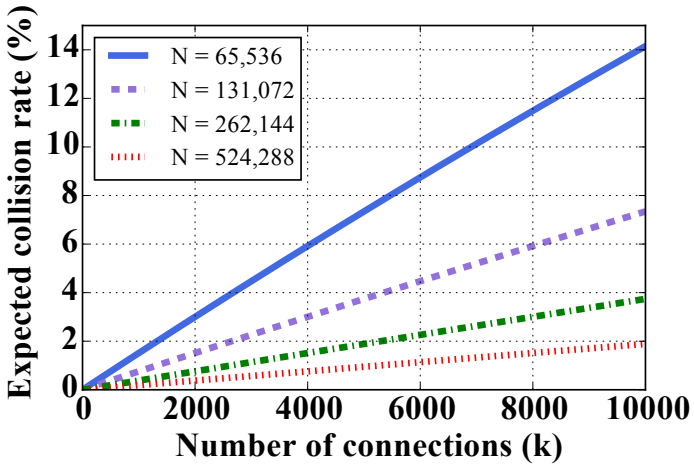


Figure 4.9: Expected collision rate vs number of flows (k) for different table sizes (N).

CPU in software: We measure the CPU overhead by connecting two servers to a single switch and starting parallel TCP flows between them and measuring CPU using `top`. The server machines have Xeon e3-1630 V3, 4 core, 3.4 GHZ processors. All flows are initially established (i.e., completed the TCP handshake) and have the average rate of 1 Mbps. Figure 4.10 shows the total CPU consumption (including live packet capturing via `libpcap` and copying packets to user space) versus the aggregate bandwidth processed. Dapper's CPU consumption is close to approaches that use near-real time polling frequency (e.g., [115] at 50ms frequency). Batch-processing can lower the CPU overhead. In addition, the overhead in system calls and memory copies caused by `libpcap` for packet capturing can be reduced if Dapper uses fast packet IO frameworks such as Netmap [83].

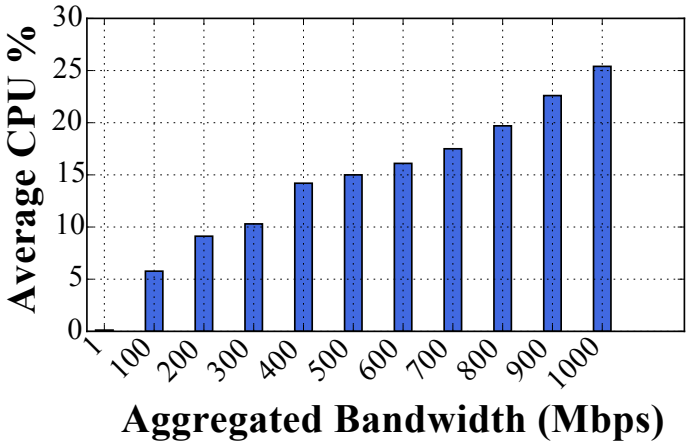


Figure 4.10: CPU per aggregate bandwidth processed.

In Figure 4.11, we quantify the CPU processing requirements for different packet types. The y-axis shows how many CPU cycles it takes after a packet is fully captured to update the flow state (excluding the packet capturing process). The first packet of a flow usually takes longer to process, because Dapper must allocate and initialize flow state and parse packet options that require extra CPU cycles. Further, the first outgoing packet is used for creating the first tuple for RTT measurement. The variations in the cycles are caused by several reasons: First, our software prototype's

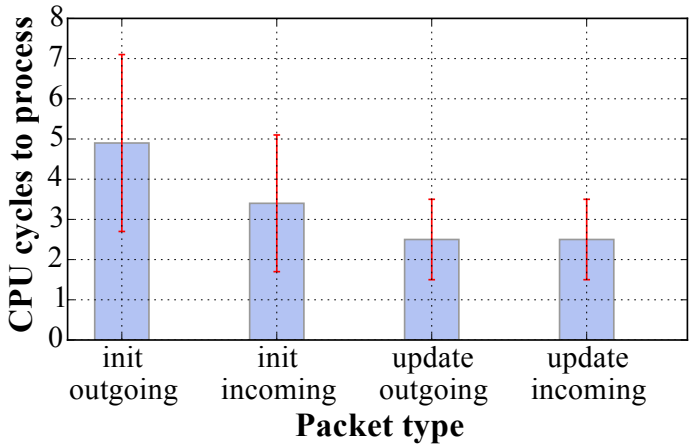


Figure 4.11: CPU cycles to update the table based on each type of packet, excluding packet capturing and handling.

flowtable is a hash table with hash-chaining, thus, in case of a collision, the flow statistics are maintained in a linked-list; incurring the extra overhead of linked-list traversal for collisions. Second, for retransmitted packets or duplicate ACKs, the flow state needs further updates. Finally, packets used for RTT measurement incur extra processing.

4.6.3 Diagnosis Accuracy

We measure the accuracy of our diagnosis method by systematically creating TCP connections with known problems and comparing the diagnosis results with the ground truth. We create a server-client connection with the client requesting a 1MB file over a link with 1Mbps bandwidth and 50ms round-trip time. We then create the following problems in the connection and repeat the experiments 100 times:

1. Sender-Limited: We emulate a resource bottleneck for the server, e.g., slow disk or busy CPU, by making the server wait for T seconds before transmitting each data packet. Higher T indicates more severe problems. We also emulate non-backlogged servers by limiting the transmitted segment sizes to less than an MSS.

2. Receiver-Limited: We create receiver-limited connections by changing socket options (using Linux's `setsockopt`) to limit the client's receive buffer size, `socket.SO_RCVBUF`.

3. Network-Limited: We use the Gilbert-Elliot model [52] to emulate micro-bursts during network congestion: a connection can be in either a *good* (no network congestion) or *bad* (network congestion) state. To emulate the bad state, we generate bursty losses at a rate of 1% to 10% for 2 seconds. We assume that losses in the good state are negligible.

For each problem shown in Table 4.2, we measure "sensitivity" and "accuracy": the *true positive rate* (TPR) is the fraction of correctly diagnosed tests, showing the sensitivity to each problem, and *diagnosis accuracy* is the fraction of time Dapper correctly classified the connection to be limited by that problem. The initial results are promising, Dapper achieves an average accuracy of 94%; Dapper's accuracy is less than 100% because the ability to detect a problem is proportional to its severity. Figure 4.12 shows an example problem where the severity changes from low (1% loss) to high (10% loss), increasing the average accuracy of diagnosis.

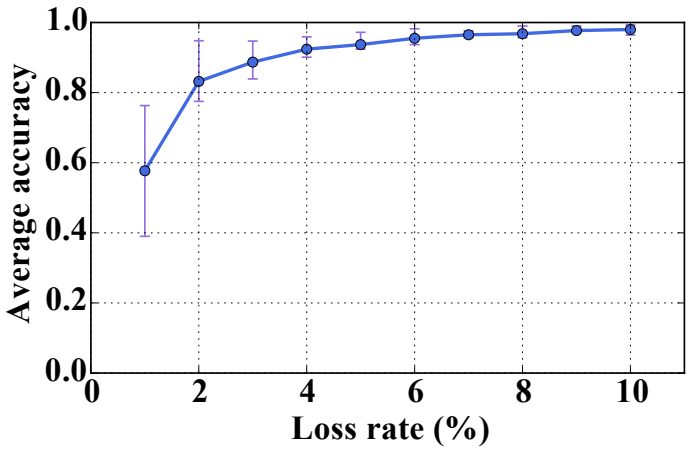


Figure 4.12: Accuracy vs severeness of problem.

Table 4.2: Dapper's Diagnosis Sensitivity and Accuracy

Problem	TPR	Avg Accuracy
Sender-Limited	98%	95%
Receiver-Limited	96%	94%
Network-Limited	94%	93%
Sender-Network-Limited	100%	95%
Receiver-Network-Limited	100%	93%

4.6.4 Analyzing CAIDA Traces

We use CAIDA traces collected on Equinix-Chicago [20] to showcase Dapper, assuming that the applications that produced this traffic could be served on a CDN. We pre-process the trace by eliminating connections with less than 10 packets in either direction. A total of 244,185 flows are considered in this study. After preprocessing, the traces are treated as a live stream of packets, with no replay. To account for the fact that the packet traces are collected from within the network, rather than the edge, we turn off inferences based on application reaction time because it cannot be reliably captured in the core of the network.

Figure 4.13 shows the CDF of the fraction of time each type of performance problems limits a connection. Since the bottlenecks in a connection may change over time, the CDF shows the normalized duration of bottlenecks. We observe that: *1.* 99% of flows spend less than 1% of their life in the no-limit state; this means that 99% of flows have at least one bottleneck in more than 99% of their lifetime. *2.* Although sender and receiver problems have similar rates, we did not use the application reaction time to detect the sender-limited connections, thus we expect the actual rate of sender-limited connections to be higher. *3.* Many connections are bottlenecked by two factors simultaneously (e.g., sender-network or receiver-network). *4.* About 90% of the connections spend some time in the network-limited state, with almost half of them being network-limited 50% of the time. [4]

[4]Note that these results are from wide-area network and the characteristics of data-center connections are different from WAN.

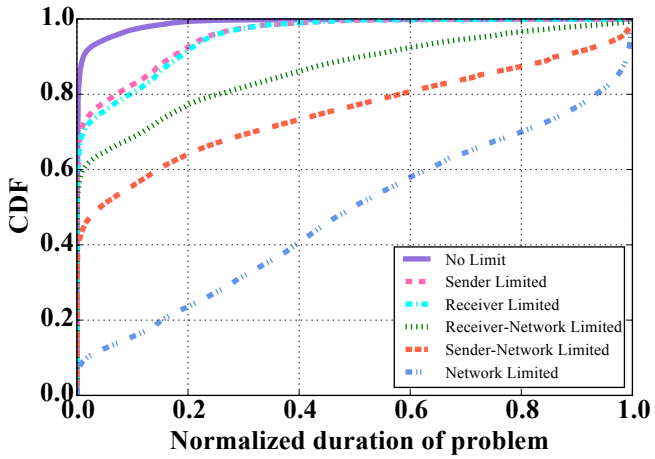


Figure 4.13: Diagnosis Results for CAIDA traces.

4.6.5 Trade-offs in Accuracy and Overhead

Using the CAIDA trace, we evaluate the impact of our space optimizations on the accuracy of our measurement. To do this, we operate our software prototype in two modes: first, trades-off space for accuracy (emulate P4) and second, measures with high accuracy and precision (ground-truth).

Limited queue size: Our measurement shows that running Dapper with an unbounded queue increases memory usage by about 9%. In a network with higher bandwidth capacity, (i.e., high bandwidth delay product), more memory will be required. In Figure 4.14, we examine the error in SRTT when the queue size is bounded to one. We observe that when the queue size is bounded, Dapper requires more samples to reduce error. Note these traces were collected in WAN, not a datacenter, hence the high RTTs.

Two-phase monitoring: Recall, two-phase monitoring offers a trade-off between accuracy of our heuristics and their memory overhead. We filter the connections to consider only those with known options, and use that as the ground-truth to compare against the midstream inferred constants. Figure 4.15 shows the error in inferring MSS and window scaling options according to Section 4.5. While the error

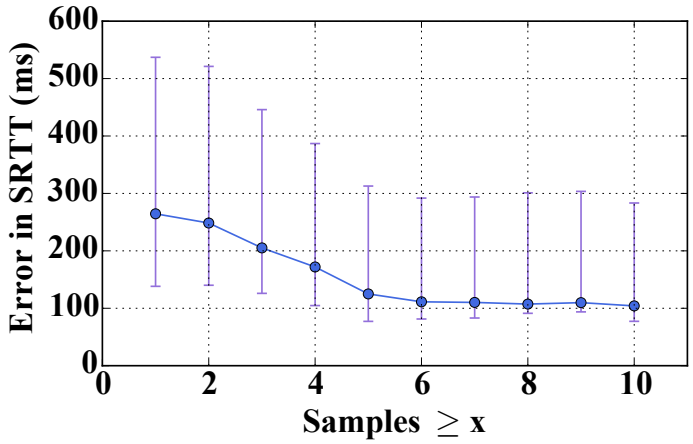


Figure 4.14: Error in inferring SRTT with queue size of 1. As more packets are exchanged, the error decreases.

in MSS decreases as more packets are inspected, the inferred window scale value still differs from the ground-truth by about 20%. This is because most of these connections are not RWND-limited, hence the flight size values used in estimating the scale (Equation 4.5) do not approach the upper-bound imposed by the receiver.

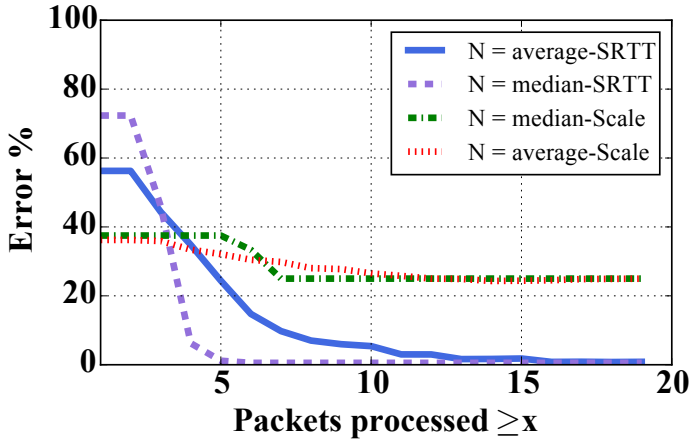


Figure 4.15: Error in inferring TCP options (MSS and wscale) midstream, the error rate decreases faster for MSS.

4.7 Discussion

In this section, we discuss the implications of implementing Dapper in hardware and our diagnosis granularity.

(1) Diagnosis Granularity: Dapper identifies the component causing the performance problem in a connection (network, sender, or receiver), but does not find the root cause within that component. As part of future work, we plan to explore domain specific inference algorithms for each entity.

(2) Hardware capabilities: A major issue in our P4 implementation arises from accessing the same registers at multiple stages, which hurts the line-rate performance at the switch. Fortunately, as discussed in Section 4.4, future versions of P4 will support the ternary operator, allowing us to effectively restrict access to registers to one stage.

(3) Hardware capacity: Switches have a limited number of registers and as discussed in Section 4.6, this capacity limitations increases the collision rate and ultimately reduces the monitoring accuracy of Dapper. To address this limitation, we propose two solutions: (i) Dapper's software equivalent (which uses `libpcap` and `BPF`) can be used in the hypervisor in public clouds. The software environment will have much more memory for storing, however, the measurement process will consume the end-host's CPU instead of the edge switch. Some of the CPU cost could be mitigated via batch-processing [70] and fast packet I/O such as netmap [83]. (ii) Sampling and triggering, as discussed in Section 4.5, can help us cope with the limited capacity.

Monitoring in the network removes packet capturing and processing overhead from the end hosts, and enables the network to adapt in real time to the diagnosis results. Implementing Dapper in the dataplane via P4 introduces limitations that a purely software solution may avoid. Yet, P4 embodies some of the capabilities at the edge (e.g. stateful programming) and provides a realistic environment. We believe that the

modest overheads are an acceptable price to pay for the flexibility of automatically embedding Dapper in a wide range of existing edge architectures.

4.8 Related Work

We have categorized the existing related work to Dapper:

Offline packet trace analysis: Several tools analyze packet traces to find performance limitation [116, 11, 97] for a known TCP variant. Some [110] store packet headers that facilitate diagnosis via running queries on headers. However, *offline* analysis makes this category unsuitable for real-time diagnosis and introduces large data-collection overhead.

Measurement in the core: Some tools use coarse-grained metrics collected on switches, e.g., 5-minute SNMP counters [109]. Such metrics are not sufficiently fine-grained to diagnose the sources of poor performance [6]; others [66] focus on coordination between network switches to ensure the full packet stream is analyzed. Instead, we rely on the "edge" for better visibility and a simpler solution.

Instrumenting the network stack: Existing end-host techniques [95, 96, 105, 115] are invasive and consume resources as they would run inside the server end-hosts. Trumpet [70], similar to Dapper, tracks packets at the hypervisor with the goal of providing a network-wide event monitoring system, but would still rely on the network operators to write diagnosis predicates.

Tomography: Network tomography infers link-level properties (e.g., loss , delay) from end-to-end measurements [18, 72, 19, 73, 90, 32] and may use linear [23, 38] or Boolean algebra [38, 106, 61] to find congested links. Our work differs from this body of work since we rely on *direct and continuous measurement* of performance instead. Tomography can be complementary to Dapper, e.g., to locate the congested link(s) among network-limited connections.

Machine learning: Recent work [4] makes the case for using machine learning (ML) in connection's performance diagnosis. Our solution differs in several aspects: first, instead on relying on ML, we leverage the fact that TCP is a well studied protocol and derive techniques based on common characteristics of TCP and its interaction with applications. Second, our solution is agnostic to application and TCP variant, while the ML would require new training for each variant.

Congestion control from the "edge": To enable new congestion control algorithms in multi-tenant clouds, [34] offers virtualized congestion control in the hypervisor. Similarly, [53] enforces congestion control in the vswitch without cooperation from the tenant VM.

4.9 Conclusion

Dapper helps CDN operators diagnose TCP connections directly in the network, at line rate, and without instrumenting the end-host servers. We use TCP domain knowledge to draw inferences about the internal state of the connections to find if a connection if limited by the sender, the network, or the receiver.

Chapter 5

Conclusion

In this thesis, we presented three systems for managing CDN performance. First, we presented CAM to allocate resources efficiently across the CDN platform by including the impact of cache misses. We use real workloads served by Akamai to choose and verify a model that best describes the relationship between the cache size and cache miss rate, and we showed that by efficient allocation of disk shares on edge servers we can enhance the performance, even with current mapping and placement algorithms in place. In CAM we model the client-perceived performance by network latency or geographical distance; however, the end-user's performance can be impacted by many other factors across the end-to-end path. To diagnose all such problems, we presented Diva.

Diva offers a chunk-based end-to-end methodology to diagnose problems within a video streaming session. Diva collects client-side information from a video player, and joins it with the server-side data and TCP connection statistics collected on the CDN servers, to build a complete and end-to-end view of the video delivery path that can diagnose many problems for the first time. Diva is the first end-to-end dataset in the video streaming literature that joins the client-side and server-side dataset and can offer ground truth in the real sources of poor performance. Diva faced some

limitations in the frequency of TCP statistics collection from the kernels, because the measurement and storage overhead of fine-grained TCP monitoring is high for an operational setting. To handle this problem, we present Dapper.

Dapper is a real-time TCP diagnostic system that runs at line-rate at the edge device, hence does not limit the edge servers or consume their resources. Dapper reconstructs the internal state of a TCP connection on the edge switch based on the incoming and outgoing packet stream, without cooperation from the OS or the application. Dapper uses the P4 language, which is among the recent frameworks for stateful programmable data planes, and offers new capabilities such as flexible parsing and registers that maintain state, which makes the detection and diagnosis of TCP performance problems possible directly in the data plane. Dapper allows the diagnosis framework to dig deeper into the network metrics and find if a TCP connection is limited by the sender, receiver, or then network.

5.1 Future Work

While we have presented our work on CDN performance management, there are many opportunities available for extending the scope of this thesis. Currently, we are working on our disk allocation algorithm in CAM to have an online version that can react to changes in cost (e.g., congestion in the network can change the latency costs), popularity distribution (e.g., certain CPs can have temporary trendy objects), and demands (e.g., peak vs. off-peak hours, or weekdays vs. weekends). We have developed a simple online greedy algorithm based on the convexity of the optimization. Our future work includes further analysis to understand (1) how well does our greedy algorithm converge to the offline optimal solution, (2) what is a good minimum size for disk allocation (i.e., unit of disk reallocation), and (3) how often should the resources be reallocated, which requires more characterization of variance within popularity

and demand. Next, we plan to expand the model we currently use in CAM to include the cache hierarchy (i.e., peers and parents). Finally, we plan to go back to the CDN join optimization problem and explore new solutions via either search optimization techniques (e.g., simulated annealing), or iterative optimization.

Bibliography

[1] A. Aggarwal, S. Savage, and T. Anderson. Understanding the performance of TCP pacing. In *IEEE INFOCOM*, 2000.

[2] Vaneet Aggarwal, Emir Halepovic, Jeffrey Pang, Shobha Venkataraman, and He Yan. Prometheus: Toward quality-of-experience estimation for mobile apps from passive network measurements. In *Workshop on Mobile Computing Systems and Applications*, 2014.

[3] Saamer Akhshabi, Ali C. Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *ACM Conference on Multimedia Systems*, 2011.

[4] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *Proc. ACM SIGCOMM*, 2016.

[5] Apache Traffic Server. `http://trafficserver.apache.org`.

[6] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proc. ACM SIGCOMM*, 2007.

[7] S. Bakiras. Approximate server selection algorithms in content distribution networks. In *IEEE International Conference on Communications*, 2005.

[8] Athula Balachandran, Vyas Sekar, Aditya Akella, and Srinivasan Seshan. Analyzing the potential benefits of CDN augmentation strategies for internet video workloads. In *Proc. ACM IMC*, 2013.

[9] A. Balamash and M. Krunz. An overview of web caching replacement algorithms. *Commun. Surveys Tuts.*, 2004.

[10] A. Barbir, B. Cain, R. Nair, and O. Spatscheck. Known content network (cn) request-routing mechanisms. RFC 3568, RFC Editor, July 2003.

[11] Paul Barford and Mark Crovella. Critical path analysis of TCP transactions. In *Proc. ACM SIGCOMM*, 2000.

[12] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM CCR*, 2014.

[13] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC*, Santa Clara, CA, 2017.

[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 2014.

[15] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proc. ACM SIGCOMM*, 2013.

[16] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. *TCP Vegas: New techniques for congestion detection and avoidance*, volume 24. ACM, 1994.

[17] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *Proc. IEEE INFOCOM*, volume 1, Mar 1999.

[18] R. Caceres, N. G. Duffield, J. Horowitz, and D. F. Towsley. Multicast-based inference of network-internal loss characteristics. *IEEE Transactions on Information Theory*, 1999.

[19] R. Caceres, N.G. Duffield, S.B. Moon, and D. Towsley. Inference of internal loss rates in the mbone. In *Proc. GLOBECOM*, 1999.

[20] The CAIDA UCSD 2013 internet traces - 2013/05/29. `http://data.caida.org/datasets/passive-2013`.

[21] Matt Calder, Ashley Flavel, Ethan Katz-Bassett, Ratul Mahajan, and Jitendra Padhye. Analyzing the performance of an anycast cdn. In *Proc. ACM IMC*, 2015.

[22] R. L. Carter and M. E. Crovella. Server selection using dynamic path characterization in wide-area networks. In *Proc. of IEEE INFOCOM*, 1997.

[23] Rui Castro, Mark Coates, Gang Liang, Robert Nowak, and Bin Yu. Network tomography: recent developments. *Statistical Science*, 2004.

[24] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *Proc. of ACM IMC*, 2007.

[25] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. Analyzing the video popularity characteristics of large-scale user generated content systems. *IEEE/ACM Transactions on Networking*, 17(5), October 2009.

[26] Changhoon Kim, Co-chair of P4 Language Design Working Group, personal communication.

[27] Hao Che, Ye Tung, and Zhijun Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7), September 2006.

[28] Umesh Chejara, Heung-Keung Chai, and Hyunjoon Cho. Performance comparison of different cache-replacement policies for video distribution in cdn. *High Speed Networks and Multimedia Communications*, 2004.

[29] Fangfei Chen, Ramesh K. Sitaraman, and Marcelo Torres. End-user mapping: Next generation request routing for content delivery. In *Proc. of ACM SIGCOMM*. ACM, 2015.

[30] Weibo Chu, Mostafa Dehghan, Don Towsley, and Zhi-Li Zhang. On allocating cache resources to content providers. In *Proce. of ACM Conference on Information-Centric Networking*, 2016.

[31] Cloudflare. `https://www.cloudflare.com`.

[32] Mark Coates, Rui Castro, Robert Nowak, Manik Gadhiok, Ryan King, and Yolanda Tsang. Maximum likelihood network topology identification from edge-based unicast measurements. In *Proc. ACM SIGMETRICS*, 2002.

[33] Edward G. Coffman, Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.

[34] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. Virtualized congestion control. In *Proc. ACM SIGCOMM*, 2016.

[35] Asit Dan and Don Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *Proc. of ACM SIGMETRICS*. ACM, 1990.

[36] G. Dimopoulos, I. Leontiadis, P. Barlet-Ros, K. Papagiannaki, and P. Steenkiste. Identifying the root cause of video streaming issues on mobile devices. In *CoNext*, 2015.

[37] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *ACM SIGCOMM*, 2011.

[38] N. Duffield. Network tomography of binary network performance characteristics. *IEEE Trans. on Information Theory*, 2006.

[39] Akamai edgescape brochure. `http://www.akamai.com/dl/brochures/edgescape.pdf`.

[40] Jairo Esteban, Steven A. Benno, Andre Beck, Yang Guo, Volker Hilt, and Ivica Rimac. Interactions between HTTP adaptive streaming and TCP. In *Workshop on Network and Operating System Support for Digital Audio and Video*, 2012.

[41] ActionScript 3.0 reference for the Adobe Flash. `http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/FileReference.html`.

[42] Michael J. Freedman, Mythili Vutukuru, Nick Feamster, and Hari Balakrishnan. Geographic locality of ip prefixes. In *IMC*, 2005.

[43] Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for LRU cache performance. *CoRR*, abs/1202.3974, 2012.

[44] Aditya Ganjam, Junchen Jiang, Xi Liu, Vyas Sekar, Faisal Siddiqi, Ion Stoica, Jibin Zhan, and Hui Zhang. C3: Internet-scale control plane for video quality optimization. In *USENIX NSDI*, 2015.

[45] Michele Garetto, Emilio Leonardi, and Valentina Martina. A unified approach to the performance analysis of caching systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 2016.

[46] Panagiotis Georgopoulos, Yehia Elkhatib, Matthew Broadbent, Mu Mu, and Nicholas Race. Towards network-wide QoE fairness using OpenFlow-assisted adaptive video streaming. In *ACM SIGCOMM Workshop on Future Human-centric Multimedia Networking*, 2013.

[47] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In *Proc. of ACM SOSR*, 2017.

[48] Mojgan Ghasemi, Partha Kanuparthy, Ahmed Mansy, Theophilus Benson, and Jennifer Rexford. Performance characterization of a commercial video streaming service. In *Proc. of ACM IMC*, 2016.

[49] Monia Ghobadi, Yuchung Cheng, Ankur Jain, and Matt Mathis. Trickle: Rate limiting YouTube video streaming. In *USENIX Annual Technical Conference*, 2012.

[50] James D. Guyton and Michael F. Schwartz. Locating nearby copies of replicated internet servers. In *Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '95, 1995.

[51] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: A new tcp-friendly high-speed tcp variant. *ACM SIGOPS Oper. Syst. Rev.*, July 2008.

[52] G. Hasslinger and O. Hohlfeld. The gilbert-elliott model for packet loss in real time services on the internet. In *Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB)*, March 2008.

[53] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *Proc. ACM SIGCOMM*, 2016.

[54] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The newreno modification to tcp's fast recovery algorithm. RFC 6582, RFC Editor, April 2012. `http://www.rfc-editor.org/rfc/rfc6582.txt`.

[55] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *ACM SIGCOMM*, 2014.

[56] V. Jacobson. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, 1988.

[57] Manish Jain and Constantinos Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput. In *Proc. ACM SIGCOMM*, 2002.

[58] Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. Shedding light on the structure of internet video quality problems in the wild. In *ACM CoNext*, 2013.

[59] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with FESTIVE. In *ACM CoNext*, 2012.

[60] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. In *Proc. of the ACM Workshop on Frontiers in Computer Communications Technology*, SIGCOMM, 1988.

[61] R.R. Kompella, J. Yates, Albert Greenberg, and A.C. Snoeren. Detection and localization of network black holes. In *IEEE INFOCOM*, 2007.

[62] Rupa Krishnan, Harsha V. Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize CDN performance. In *IMC*, 2009.

[63] S. Shunmuga Krishnan and Ramesh K. Sitaraman. Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. In *Proc. ACM IMC*, 2012.

[64] It's latency, stupid. https://rescomp.stanford.edu/~cheshire/rants/Latency.html.

[65] Xi Liu, Florin Dobrian, Henry Milner, Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. A case for a coordinated Internet video control plane. In *ACM SIGCOMM*, 2012.

[66] Xuemei Liu, Meral Shirazipour, Minlan Yu, and Ying Zhang. Mozart: Temporal coordination of measurement. In *Proc. ACM SOSR*, 2016.

[67] Bruce M. Maggs and Ramesh K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM CCR*, July 2015.

[68] MATLAB. version 9.10.441655 (r2016b), 2016.

[69] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, June 1970.

[70] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proc. of ACM SIGCOMM*, 2016.

[71] T. R. Gopalakrishnan Nair and P. Jayarekha. A rank based replacement policy for multimedia server cache using zipf-like law. *CoRR*, abs/1003.4062, 2010.

[72] Hung X. Nguyen and Patrick Thiran. Binary versus analogue path monitoring in IP networks. In *Proc. PAM*, 2005.

[73] Hung X. Nguyen and Patrick Thiran. Network loss inference with second order statistics of end-to-end flows. In *Proc. ACM IMC*, 2007.

[74] Nmap. http://www.nmap.org.

[75] Frank Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Technical report, Lawrence Berkeley Lab., CA (USA), 1981.

[76] The P4 language specification, version 1.0.2. http://p4.org/wp-content/uploads/2015/04/p4-latest.pdf.

[77] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), 2000. Obsoleted by RFC 6298.

[78] Louis Plissonneau and Ernst Biersack. A longitudinal view of HTTP video streaming performance. In *Multimedia Systems Conference*, 2012.

[79] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, December 2003.

[80] Ingmar Poese, Steve Uhlig, Mohamed Ali Kaafar, Benoit Donnet, and Bamba Gueye. IP geolocation databases: Unreliable? *ACM SIGCOMM CCR*, 2011.

[81] David M. W. Powers. Applications and explanations of zipf's law. In *Proc. of the Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning*. Association for Computational Linguistics, 1998.

[82] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proc. of Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, 2002.

[83] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *USENIX ATC*, 2012.

[84] Elisha J. Rosensweig, Jim Kurose, and Don Towsley. Approximate models for general cache networks. In *Proc. of IEEE INFOCOM*, 2010.

[85] John P. Rula and Fabian E. Bustamante. Behind the curtain: Cellular dns and content replica selection. In *Proc. ACM IMC*, 2014.

[86] Sandvine: Global Internet phenomena report 2015. `https://www.sandvine.com/trends/global-internet-phenomena/`.

[87] S. Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. In *IEEE INFOCOM*, 1999.

[88] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *Proc. ACM SIGCOMM*, 2016.

[89] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of dns-based server selection. In *Proc. of IEEE INFOCOM. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, 2001.

[90] Y. Shavitt, Xiaodong Sun, A. Wool, and B. Yener. Computing the unmeasured: An algebraic approach to internet mapping. In *Proc. IEEE INFOCOMM*, 2001.

[91] Shan-Hsiang Shen and Aditya Akella. An information-aware qoe-centric mobile video cache. In *Proc. of MobiCom*, 2013.

[92] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proc. of ACM SIGCOMM*, 2016.

[93] Han Hee Song, Zihui Ge, Ajay Mahimkar, Jia Wang, Jennifer Yates, Yin Zhang, Andrea Basso, and Min Chen. Q-score: Proactive service quality assessment in a large IPTV system. In *IMC*, 2011.

[94] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proc. of ACM SIGCOMM HotSDN*, 2013.

[95] Peng Sun, Minlan Yu, Michael J. Freedman, and Jennifer Rexford. Identifying performance bottlenecks in CDNs through TCP-level monitoring. In *ACM SIGCOMM Workshop on Measurements Up the Stack*, 2011.

[96] Peng Sun, Minlan Yu, Michael J. Freedman, Jennifer Rexford, and David Walker. HONE: Joint host-network traffic management in software-defined networks. *Journal of Network and Service Management*, 2015.

[97] tcptrace. `http://tcptrace.org/`.

[98] Guibin Tian and Yong Liu. Towards agile and smooth video adaptation in dynamic HTTP streaming. In *CoNext*, 2012.

[99] Open read retry timer. `https://docs.trafficserver.apache.org/en/4.2.x/admin/http-proxy-caching.en.html#open-read-retry-timeout`.

[100] Tom Tofigh and Nic Viljoen. Dynamic analytics for programmable NICs utilizing p4 - identification and custom tagging of elastic telecoms traffic. `http://p4.org/wp-content/uploads/2016/06/P4-Poster-Netronome-ATT.pdf`.

[101] Ruben Torres, Alessandro Finamore, Jin Ryong Kim, Marco Mellia, Maurizio M. Munafo, and Sanjay Rao. Dissecting video server selection strategies in the YouTube CDN. In *International Conference on Distributed Computing Systems*, 2011.

[102] Anna Saro Vijendran and S. Thavamani. Survey of caching and replica placement algorithm for content distribution in peer to peer overlay networks. In *Proc. International Conference on Computational Science, Engineering and Information Technology*, 2012.

[103] Jia Wang. A survey of web caching schemes for the internet. *ACM SIGCOMM CCR.*, 29(5), October 1999.

[104] Nicholas Weaver, Christian Kreibich, Martin Dam, and Vern Paxson. Here be web proxies. In *PAM*, 2014.

[105] The Web10G project. `http://www.web10g.org`.

[106] Wei Wei, Bing Wang, Don Towsley, and Jim Kurose. Model-based identification of dominant congested links. In *Proc. of ACM IMC*, 2003.

[107] Patrick Wendell, Joe Wenjie Jiang, Michael J. Freedman, and Jennifer Rexford. Donar: Decentralized server selection for cloud services. In *Proc. ACM SIGCOMM*, 2010.

[108] Kin-Yeung Wong. Web cache replacement policies: A pragmatic approach. *IEEE Network*, Jan 2006.

[109] L4Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. *ACM SIGCOMM CCR*, 2012.

[110] Wenfei Wu, Guohui Wang, Aditya Akella, and Anees Shaikh. Virtual network diagnosis as a service. In *Symposium on Cloud Computing*, 2013.

[111] Xing Xu, Yurong Jiang, Tobias Flach, Ethan Katz-Bassett, David Choffnes, and Ramesh Govindan. Investigating Transparent Web Proxies in Cellular Networks. In *Proc. of PAM*, 2015.

[112] Hao Yin, Xuening Liu, Feng Qiu, Ning Xia, Chuang Lin, Hui Zhang, Vyas Sekar, and Geyong Min. Inside the bird's nest: Measurements of large-scale live VoD from the 2008 olympics. In *Proc. ACM IMC*, 2009.

[113] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *Proc. ACM SIGCOMM*, 2015.

[114] Youtube statistics. `https://www.youtube.com/yt/press/statistics.html`.

[115] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *Proc. NSDI*, 2011.

[116] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of Internet flow rates. In *Proc. ACM SIGCOMM*, 2002.