



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Artificial Intelligence

Media over QUIC Relay Topology Optimization

Scientific Students' Association Report

Author:

Zoltán Szatmáry
Balázs Szenczy

Advisor:

Dr. Tamás Lévai
Dr. Gábor Rétvári

2024

Contents

Kivonat	i
Abstract	ii
1 Introduction	1
2 Background and Related Work	6
2.1 Media over QUIC	7
3 Modeling and Optimizing the Relay Topology	10
3.1 Problem Formulation	11
3.1.1 Problem Formulation for a single track	11
3.1.2 Problem Formulation for multiple tracks	13
3.2 Problem Classification	14
3.3 Heuristics	16
3.3.1 Complexity Analysis	18
3.4 Summary	18
4 Implementation	19
4.1 Architecture	19
4.2 Optimizer	20
4.3 Testbed	20
5 Evaluation	22
5.1 Benchmark	22
5.2 Testbed	25
6 Conclusion	27
6.1 Future Work	27
Bibliography	29

Kivonat

Napjainkban az Internet forgalom több mint 80%-a médiatartalom, amelynek jelentős része késleltetés-érzékeny (pl. videokonferencia, live streaming, cloud gaming). Az ilyen alacsony-késleltetésű média átvitelére manapság a WebRTC szolgál. Azonban a WebRTC architektúra komplex, nehezen skálázódik és illeszkedik a modern távközlési rendszerekhez (pl. számítási felhő). E probléma megoldására hozta létre az IETF a Media over QUIC (MoQ) kezdeményezést, ami QUIC protokoll felett valósítja meg a média/adat átvitelét. Számos különböző átviteli modelleket és követelményekkel rendelkező alkalmazást támogat. Ehhez a MoQ a tartalomelosztó hálózatok architektúráját követi. Elemei: küldő, továbbító, fogadó. A küldő hozza létre és továbbítja a médiatartalmat a továbbítónak. A fogadó továbbítóhoz csatlakozik a médiatartalom eléréséhez. A továbbítók topológiába rendezhetők. Ezen architektúrának köszönhetően képes médiatartalmak hatékony továbbítására, a jövő meghatározó médiaelosztó megoldásává válhat.

Azonban a jelenlegi implementációk nem támogatják a továbbító hálózatban a médiatartalmak útvonalának kiválasztását. Ez azonban kulcseleme a kis késleltetésű és alacsony költségű továbbításnak nagy kiterjedésű hálózatban. Jelenleg a továbbítók teljes gráfot alkotnak, a küldő által használt továbbítóhoz a fogadó közvetlenül csatlakozik. Ez a megoldás bár a legkisebb végpontok közötti késleltetést nyújtja, nagy hálózati terheléssel jár és jelentősen korlátozza a skálázódást.

Ebben a dolgozatban a médiatartalmak útvonalválasztására adunk újszerű megoldást, amivel csökkenthető a MoQ továbbító topológia üzemeltetési költsége, a késleltetés korlátok betartása mellett. Megfogalmazzuk a költséghatékony és késleltetésérzékeny média útvonalválasztást matematikai problémaként, meghatározzuk a komplexitását, optimális és heurisztikus algoritmusokat adunk a megoldására. Továbbá integráljuk a megoldásunkat egy elterjedt MoQ implementációba, és létrehozunk egy programozható realisztikus mérési környezetet. Megközelítésünk hatékonyságát részletes elméleti kiértékelés és a mérési környezetben végrehajtott mérések igazolják. Különböző szintetikus és valószerű topológián végzett méréseink eredményei jelentős költségcsökkenést mutatnak a késleltetés korlátok betartása mellett.

Abstract

According to recent studies, over 80% of global Internet traffic is media. Much of this is latency-sensitive (e.g., videoconferencing, live streaming, cloud gaming, etc.). The state-of-the-art solution for low-latency media transmission over the Internet is WebRTC. WebRTC has a complex architecture that is hard to scale and adapt to modern telecommunication systems such as clouds. To overcome this limitation, the IETF recently introduced the Media over QUIC (MoQ) initiative, which enables the transport of media/data streams over the QUIC protocol. Thanks to its generality, it supports multiple use cases with different transport models and requirements. MoQ employs an architecture akin to Content-Delivery Networks, comprising three main elements: publishers, subscribers, and relays. Publishers generate and send media tracks to relays. Subscribers join relays to access specific media tracks. Relays can form a delivery network to transmit media. This architecture ensures that media can be distributed at scale without compromising performance, making MoQ a crucial component for modern media delivery systems.

Unfortunately, current implementations lack support for media routing, which is necessary for low-latency and low-cost transmission at scale. Relays create a fully meshed network to transmit media streams from the publisher's to the subscriber's relay. Although this yields the best end-to-end latency, it results in extremely high network utilization and limits scalability.

In this work, we present a novel approach to enforce media routing and lower the overall costs of operating a MoQ relay topology while satisfying delay service level objectives. We formulate the cost-efficient and delay-aware media routing as a mathematical program, characterize the computational complexity, and propose optimal and heuristic algorithms to solve it. We integrate our approach with a widely-used MoQ implementation. We also introduce a programmable realistic testbed to demonstrate our work. The viability of our approach is confirmed in comprehensive numerical analysis and experiments in the testbed. Our evaluation of different synthetic and realistic topologies shows improvements in cost-efficiency while keeping all delay constraints satisfied.

Chapter 1

Introduction

We use media-based solutions in almost every aspect of our digital lives from voice and video calls to live streaming, cloud gaming, and beyond. This contributes significantly to the network traffic on today's large-scale networks. Surveys show that media traffic puts more than 80% of all network traffic [1], although some surveys claim that this number is closer to 60% [2], therefore, it is expected that the cost-efficient distribution of massive-scale media is a key goal of current and future networks.

A major challenge in media transmission is that many use cases demand not only the ability to download or upload media eventually but require it to happen in real time. Real-time communication is a type of communication in which information exchange occurs with little to no delay involved. Physical constraints make it impossible to completely eliminate latency; however, most communication applications do not impose such strict restrictions. For example, during voice calls, people can generally tolerate some delay, as long as the communication appears continuous and uninterrupted.

Traditionally, low-latency real-time communications used to be done over circuit-switched networks, where Quality of Service (QoS) was guaranteed. Despite the undeniable advantage of circuit-switched networks in guaranteed QoS, they are losing in favor of packet-switched networks. Packet-switched networks provide best-effort delivery, meaning that there is absolutely no guarantee for a data packet to be delivered (let alone in-time delivery); in turn, they are more flexible and cheaper to maintain. However, achieving real-time communication on large-scale, public, packet-switched networks such as the Internet is a particularly challenging task due to the lack of control over the intermediary devices involved during the transmission of packets from source to destination.

As mentioned above, there are many use cases for media transmission over the Internet; however, each of them can have differences in terms of bandwidth usage, delay requirements, and the typical communication pattern in which it is involved. In the following paragraphs, we summarize some of these characteristics for three popular use cases.

Video calls and videoconferencing stand out as a crucial component for remote communication in our daily lives. In this accelerated world, every business relies on videoconferencing technologies for various things (e.g., to conduct meetings, interviews, etc.). There is also clearly a strong demand for video calls in education. During the pandemic and the lockdown, every educational institution had to switch to distance learning, in response to which many lessons were conducted on video conferencing platforms such as Google Meet. In a regular video conference, every member in the conference can talk, enable their webcam, share their screen, etc., which makes this type of communication many-to-many. While screen sharing or the video feed of a webcam may tolerate some amount of delay,

when combined with audio, it becomes crucial to make them stay in sync, and therefore, to adjust the delay bound accordingly (i.e., to the smallest delay required by the audio and video tracks), otherwise the Quality of Experience (QoE) will degrade significantly.

Live streaming is another popular real-time (or near-real-time) media application that is used by hundreds of millions of people around the world [3]. Normally, a live streaming scenario consists of one publisher and many subscribers (i.e., viewers of the live content), thus it follows a one-to-many pattern of communications. There are exceptions to this, for example, Twitch has a feature called Stream Together [4], which allows multiple streamers to publish a live stream together, but from a technical perspective, it is still closer to the one-to-many pattern described. Live streaming, despite the word “live” in its name, usually has relaxed delay requirements compared to voice or video calls. This is mainly due to the lack of real-time interactions. However, there are situations where the delay of a stream gets more importance, and even a few seconds of delay makes the viewing experience degrade. Like it was experienced at the latest Superbowl broadcast [5].

Cloud gaming is another promising area where we expect to see improvements. It is a segment of the video game industry in which the game runs on a remote server instead of the user’s device, making it a thin-client architecture. The game’s controls (i.e., the events triggered by the user’s input devices) are sent from the user to the remote server, and in turn, the server sends back the rendered frames and the audio samples that should be played out at the user. The advantage of this is that it enables virtually every hardware with decent Internet connectivity to be used to play games on it. Since users seek immediate feedback for their interactions in real-time, this is yet another area, where end-to-end delays higher than a few dozens of milliseconds are not tolerable. Generally, due to the nature of the most popular games on the market, cloud gaming implies a one-to-one communication pattern; however, a niche market where cloud gaming is also an example of many-to-one communications is the world of split-screen multiplayer and party games. In such games, every player participating sees the exact same content on their screen (and hears the same audio), but can control their avatars individually. We believe that the latter use case could benefit substantially from improvements in cost-efficient content distribution.

The latency requirements for media transmission can largely vary depending on the intent of the communication. For example, voice and video calls are considered real-time services, while live streaming is only quasi-real-time. This is related to the level of interactivity that the specific application needs. As shown in Figure 1.1, in determining the effects of absolute delay by the E-model, people are very sensitive to delays longer than a few hundred milliseconds. Because of that, the International Telecommunication Union Telecommunication Standardization Sector (ITU-T) recommends keeping the latency for audio transmission **below 150 ms** to make interactivity appear transparent. For one-way video use cases, such as live streaming, **400 ms** is advisable.

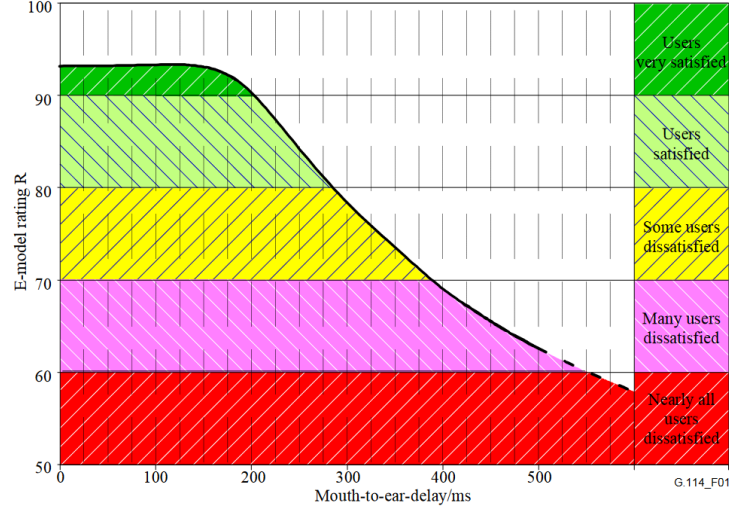


Figure 1.1: User satisfaction at different delay values according to ITU-T G.114 industry recommendation [6]

Currently, most media transmission with strict latency requirements over the Internet is performed over WebRTC. Initially, WebRTC started as a technology stack built for video conferences, it has grown out, and nowadays many other applications use it for real-time peer-to-peer communications outside of video conferencing, as an important benefit of using WebRTC for anything outside of video conferencing is its ubiquity and the fact that it is supported by all major browsers. Architecturally, it is considered to be a monolith due to its Voice over IP (VoIP) legacy, and it is generally quite difficult to adapt it to novel infocommunication systems, such as the cloud, due to its complex architecture.

To solve the problems with WebRTC, the Internet Engineering Task Force (IETF) has initiated a standardization effort through its Media over QUIC working group, focused on developing a streamlined, low-latency media delivery solution. Media over QUIC (MoQ) is a new protocol being developed by this working group that aims to solve the challenges posed by WebRTC’s architecture and monolithic nature. In this paper, we propose a method to address this issue. MoQ is designed to support a variety of different use cases, including those mentioned above (video conferencing, live streaming, and cloud gaming). The draft maintains a straightforward approach by delineating key aspects, including publisher, subscriber clients, and relays. These relays are available to clients and have the ability to connect with one another. The relay network enables high scalability potential for systems using MoQ. It is also worth pointing out that the creator of MoQ, Luke Curley, was motivated by the problems he had seen with protocols such as WebRTC supporting live media transmission during his time at Twitch, a company specialized for live video streaming [7]; he focuses on the live media use case and maintains a demo for live streaming with MoQ on his website [8]. Likewise, there also exists a pilot for using Media over QUIC based distribution in cloud gaming [9]. Although MoQ is rapidly advanced by the working group, it still lacks certain critical features, such as the media routing capabilities necessary for deployment in intricate cloud network topologies.

To demonstrate the need of MoQ media routing, we present a simple network called *small_topo*, illustrated in Figure 1.2. Hosts are represented by text boxes, with the publisher’s relay shown in blue, and the subscribers’ relays in red. The last-mile connection to the client’s first relay is not displayed, as we have minimal control over it. Overlay connections are indicated by dashed lines, while straight lines represent underlay connections.

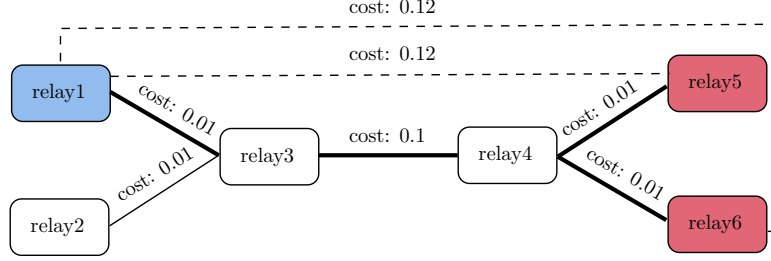


Figure 1.2: The *small_topo* topology, with the media routed lower cost links highlighted

We use overlay links to demonstrate IP routing, utilizing the same underlay links between two hosts but using Layer 3 routing. In contrast, using the underlay links in this model, will route the media traffic differentiation the protocol-defined tracks and groups, in Layer 7. Every host is interconnected through overlay links, but for simplicity, only essential dashed lines are shown to enhance clarity. Media routing in Layer 7 can effectively utilize the underlay links. Although the financial benefit of media routing is not outstanding here, it is evident even in this small instance. In theory with IP routing, using the two overlay links, the dashed ones for the two subscribers result in a cost total of 0.24. In contrast, with media routing, data is sent only once until relay4, where it is subsequently divided for the two subscribers, yielding a total cost of 0.13. In this way, we can save 46% of the costs.

Originally, the Internet was designed to be used for one-to-one communication, also known as unicasting. It works well for many use cases; however, in cases where the same content needs to be distributed to multiple peers, this poses unnecessary duplications and thus increased utilization on the network. An early attempt at solving this came with IP Multicast. Although it has become a go-to solution for efficient media distribution on managed networks (e.g., for use in IPTV), due to the lack of global support for IP Multicast protocols, it is unavailable for use on the public Internet [10]. Consequently, on the Internet, communication remains constrained to unicasting. Recognizing these constraints, Application Layer Multicast (ALM) emerged as a practical Layer 7 approach to bring multicast capabilities to networks without native IP Multicast support at Layer 3 [11]. We argue that media routing solutions based on ALM provide the necessary scalability for media distribution systems on large-scale networks. In addition, the ideal solution also needs to take account of the end-to-end delays measured from publisher to subscriber, in order to provide QoS. Our solution improves the state-of-the-art MoQ implementation by extending it with media routing capabilities to provide ALM.

Next, we overview the possible solutions for MoQ media routing. Intuitively, this consists of two extremes: one with extremely low latency (provided by the state-of-the-art (SOTA) *no media routing* solution) and the other with the lowest operational costs (provided by a minimum spanning tree (MST) based optimization strategy that does not consider any delay bounds). In most cases neither of them suits our goals and that a balance has to be found between the two. In Figure 1.3 we show a sample visualization of this theoretical delay-cost spectrum. There is a trade-off: We have to compromise on one parameter to gain on the other. The ideal solution to the problem should be flexible. Minimizing the costs by default is the main goal; however, when the delay bound could not be satisfied via that link, it can adapt its strategy by falling back to using another one that may not be cost-optimal otherwise, but conforms the E2E delay bound. For instance, consider a network where the publisher’s relay has high-cost links to each subscriber’s relays with

Chapter 2

Background and Related Work

There have been multiple solutions for efficiently distributing media content across geographically dispersed users, optimizing both speed and bandwidth usage [12]. Media over QUIC (MoQ) offers a modern approach to media transmission, enhancing scalability and reducing latency by utilizing relays that streamline content distribution, making it highly effective for dynamic and large-scale media applications.

In this chapter, we introduce two related fields from the field of video distribution:

- **Content Delivery Networks** (CDNs) used in Internet TV solutions (live video or video on demand) with over-the-top (OTT) distribution through public networks.
- **Multicasting** used in live video distribution through managed private networks.

We argue that our problem with the cost-efficient and low-latency distribution of content in MoQ is closely related to other (usually video-based) media distribution problems. A ubiquitous problem that arises in virtually every content delivery system is that the centrality of the data has to be given up in order to provide better latency to users. Content Delivery Networks (CDNs) were created as a workaround to this problem. CDNs work by distributing content across multiple servers at several physical locations instead of relying on a single centralized server, so that users can get the content from the nearest or otherwise most optimal server. It can offer several benefits. Notable ones include caching, cost-efficient operation (due to only having to move large volumes of data once for every CDN), and low-latency distribution of on-demand content. Figure 2.1 shows the two scenarios. On the left, a single server serves every client, regardless of their physical location. In contrast, on the right, each client is served by the proxy server that is closest to the client in a way (e.g., geographically).

From the content provider’s perspective, a CDN can be either private or third-party. A private CDN is owned by the provider itself and is designed to fit the specific needs of the business. Netflix’s Open Connect content delivery network [13] specialized for on-demand video streaming is an example of such a private CDN. In comparison, a third-party CDN can be used by many parties to host a variety of different things, from static files of a website to images, videos, or audio files. Akamai CDN, Cloudflare CDN and Amazon CloudFront are among the largest third-party CDNs as of 2024 [14].

Multicasting is the simultaneous transmission of the same data to one or more destinations [15]. A common application of multicasting is media delivery, for example, in Internet Protocol Television (IPTV) [16]. Multicasting has no direct support in the Internet Protocol (IP); therefore, implementing Layer 3 multicasting in IP-based networks requires specialized protocols such as Protocol Independent Multicast (PIM) and Internet

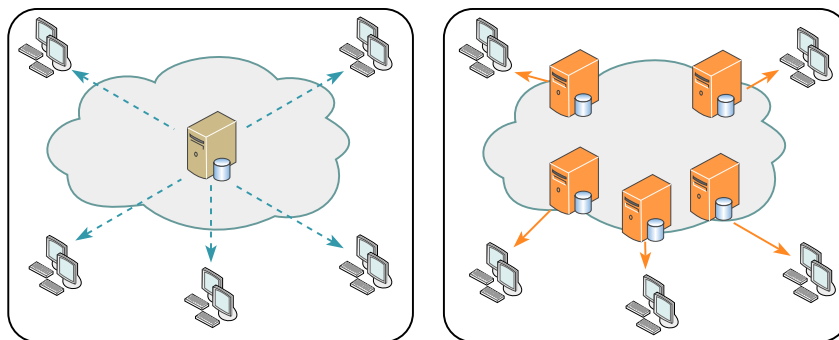


Figure 2.1: Distribution without a CDN (left) and with a CDN (right) [17]

Group Management Protocol (IGMP). This means that multicasting is mostly limited to private, managed networks with proper equipment supporting the appropriate protocols and configured as such.

Some applications, such as multiplayer games or live streaming of sporting events to numerous locations, require sending packets to multiple recipients. The transmission of individual packets to each receiver becomes costly if the group is large. In contrast, broadcasting packets to a group of, for example, 1000 machines in a million-node network is inefficient because many receivers may either be uninterested in the message or, worse, should not see it. Therefore, a method is needed to send messages to specific and well-defined groups that are large in number but small relative to the entire network. Mostly multicast routing is used for media such as VoIP, live streaming, and IPTV.

2.1 Media over QUIC

Moving away from multicast at the data link layer, other solutions exist. QUIC, a novel transport layer protocol, is built on top of UDP [18], incorporating numerous features of TCP and adding new enhancements. Like TCP, it ensures reliability while integrating built-in encryption. QUIC supports zero round-trip connection initiation, leveraging TLS1.3 to facilitate the concurrent transmission of connection requests, encryption negotiations, and initial data. A significant concern with TCP, namely head-of-line blocking, is addressed in QUIC through its multistream capabilities. This means that a retransmission in one stream does not disrupt others. To improve Web performance and manage simpler requests, HTTP3 - built on QUIC - is widely supported by modern browsers¹. For low-latency, bidirectional communication using HTTP3, WebTransport is a viable solution [19]. With a versatile API, integrated into various browsers², it offers a convenient choice for web applications. Building on top of these one can reach a protocol that is deployable in a less complex way than current WebRTC solutions.

The fast establishment of connections and the capability for multiple streams make QUIC an ideal foundation for developing a media-focused protocol. The primary reason why

¹HTTP3 browser support: <https://caniuse.com/http3>

²WebTransport browser support: <https://caniuse.com/webtransport>

WebTransport is necessary, is because web browsers have a comprehensive API for it. Additionally, media is predominantly accessed through web browsers rather than native applications. The MoQ framework enables the mapping of media formats and files with distinct protocol elements. Media can be structured as a single stream, named as track-namespace; here, each video or audio track is linked to a MoQ entity known as a track. These tracks can be grouped to signal start and end markers of joining points for clients. From an architectural standpoint, the protocol discerns between publishers and subscribers who share data. Relays serve as the intermediaries, facilitating this exchange by collecting data from publishers and distributing it to subscribers.

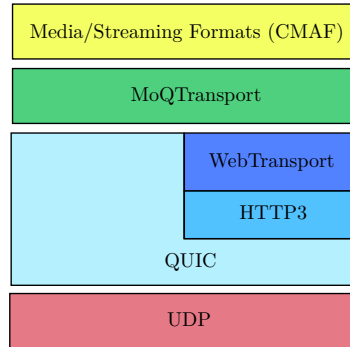


Figure 2.2: Media over QUIC protocol stack

The used layers are illustrated in Figure 2.2. With QUIC or WebTransport, as previously discussed, MoQ is capable of directly managing the media payload. The predominant implementation deals with sending and receiving CMAF segments, while the IETF Working Group is also drafting a dedicated container format named Low Overhead Container (LOC) [20].

The IETF drafts outline guidelines for Media over QUIC, which are demonstrated in several implementations. Notable examples include Facebook’s Moxxygen relay and its associated clients, with the most prominent being moq-rs. Moq-rs stands as an open-source, de-facto implementation of the protocol, written in Rust. The project comprises multiple components spread across different components. We used the following elements in this project:

- moq-relay** A server which can be connected by the clients and accept incoming data from publishers and send it out to subscribers.
- moq-pub** A publishing client that accepts media from the standard output and sends it to a remote server.
- moq-sub** A subscriber client that gets media from a relay and passes it to the standard output.
- moq-api** An HTTP API server that stores the origin for each broadcast, backed by redis. It is used when a relay does not have the publishing client at hand.
- moq-gst** In short, GStreamer is a very complex library for constructing graphs of media-handling components, with many features. This project contains a moq-sink element that can be added to GStreamer media-handling pipelines to publish media to MoQ.

By default, all relays in a network have one common API server to which they connect. All relays are connected in a full mesh topology so they are expecting to get the publishing relay in one hop from the common API server.

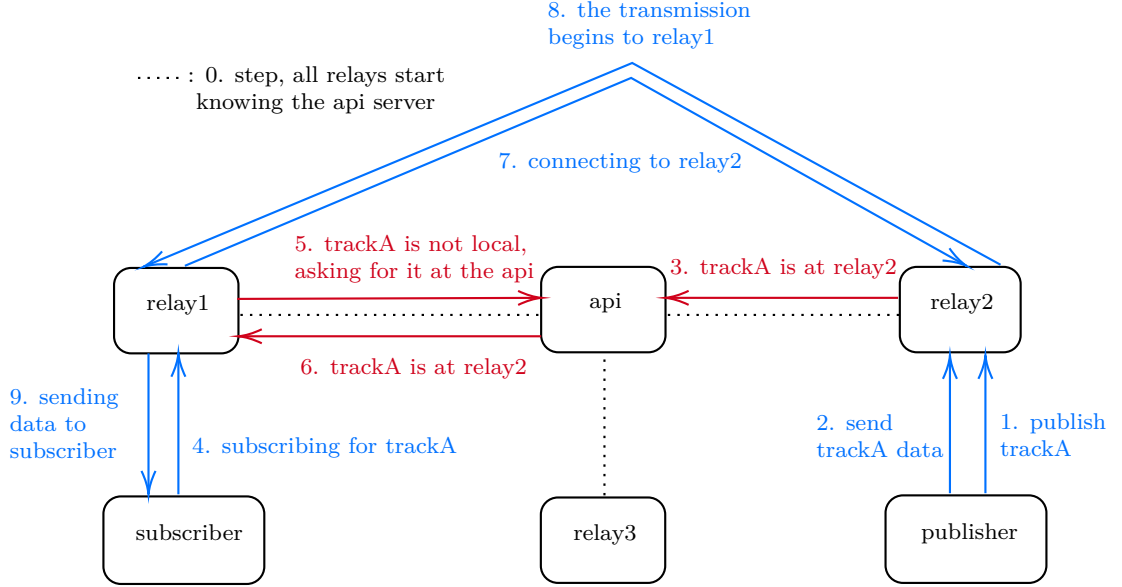


Figure 2.3: moq-rs message flow: A publisher transmits a track to a subscriber through a relay network

In Figure 2.3, there is a simplified topology where one can follow through a basic transmission. All relays and the API server are integrated into a single network or cluster. The dotted lines do not mean that there is an explicit connection between the relays and the API server but that they know the address and can connect to the server. The red steps represent HTTP2 requests and responses to and from the API server. The blue arrows indicate MoQ connections, signifying that every such line represents a MoQ connection where data is exchanged. This representation implies that the underlying QUIC or WebTransport connections are established over which MoQ messages are transmitted.

The following steps can be seen. The publishing client prepares the data locally and then sends out a unique identifier, track name, which will serve as the broadcast's identifier. Upon receiving this publishing message, the relay sends a notification to the API server indicating that if a client is looking for this specific track name, it can be accessed through it. An entry is created in the API server, the key is the specified uid, and the value is a simple JSON containing the relay's host information. When a subscribing client wants to access the specified stream using that track name, it sends a request to its relay. The relay then searches the track locally. If it cannot find it, it will query the API server. The API server receives the request for the specific track name and responds with the address of the publisher's relay. The subscriber's relay connects to the publisher's relay, then the transmission starts.

As we saw, moq-api responds with the publisher's relay when a subscriber's relay requests a specific track. This poses a problem for implementing media routing. Hence, in Section 4 we show how we adjusted the API and relays to facilitate media forwarding among multiple relays.

Chapter 3

Modeling and Optimizing the Relay Topology

Media-heavy applications and real-time services are increasingly common nowadays, hence optimizing network topologies has become essential for achieving efficient communication while ensuring service level guarantees. Media routing is essential for providing such services at large-scale (e.g., due to geodistribution); however, it needs certain extensions to fit the world of Layer 7. Similarly to what is seen at Layer 3, the MoQ media routing at Layer 7 also focuses on the control plane functions while leaving forwarding decisions (which are based on the information provided by the routing) to the relays. Media routing essentially defines topologies for each track that can tell the relays what their next-hop relay is for that specific track (i.e., another relay to ask the track from).

This chapter outlines the formalization of our problem and then describes various optimization methods (e.g., heuristic and ILP-based) that can be used by a media routing algorithm to determine the optimal topology, comparing their benefits and potential trade-offs.

Our take on modeling network topologies formed by relays resembles key characteristics of CDNs. We define relay topologies (or relay networks) as special graphs where the nodes represent relays and the edges or links (note that we will refer to them as *links* from now on) represent logical interconnections between each two nodes of the graph. The logical nature of links comes from the fact that relay topologies are generally modeled on higher abstraction levels, specifically, in the application layer (also referred to as Layer 7 or L7 for short) of the Open Systems Interconnection (OSI) reference model [21]. We can leverage the end-to-end connectivity - provided by lower-level layers - in Layer 4 and above. Therefore, when modeling links between relay nodes, an actual physical path (e.g., the shortest path) going through a series of routers and other network devices between two nodes can be relaxed into a single logical link.

In our model, we abstract the fact that in a real-life Media over QUIC topology, publishers and subscribers are not actually relays, but simple end devices that connect to (possibly to the nearest) relays. We can do this without loss of generality, since it is not a significant difference to real life. For this reason, we will refer to publishers and subscribers as their relays, in contrast to the terminology used in the IETF draft and in Section 2.1.

The distributed content (e.g., the media) in our model is organized into tracks. A track represents a content originating at exactly one publisher, that is to be distributed to zero or more subscribers. A relay topology is not dedicated to a single track; in fact, with a little

modification, our proposed solution is capable of handling multiple tracks. However, when determining the optimal topology for a certain track, we only account for the potential traffic generated by the distribution of that track. This means that even when we have multiple tracks, each of those will have a topology optimized independently of each other in complete isolation. This relaxation is acceptable under our conditions, due to the fact that we are working with logical links, of which there may be many possible mappings to disjoint paths of physical links; therefore, two independent tracks sharing the same logical link will likely not result in bandwidth constraints.

A key goal of a topology optimizer is to calculate the most cost-efficient but delay-constraint-satisfying topology for the distribution network, while keeping the end-to-end (E2E) latency measured between the communicating parties (i.e., for each publisher and its subscribers) under a predetermined delay budget (or bound).

In subsequent sections, we will formulate the problem incrementally. We begin by showing a *Linear Programming* for a single track without delay constraints, then refine it as an *Integer Linear Programming* to also account for the delay constraints. Next, we generalize the ideas for multiple tracks without delay constraints. Finally, we wrap up by classifying the problem and concluding its complexity.

3.1 Problem Formulation

3.1.1 Problem Formulation for a single track

The ultimate goal of the optimization is to have an interconnection (formed by relays) between the given tracks' publishers and subscribers, such that the total operational costs are minimal, while the delay requirements are satisfied. Formally, the input of the optimization is a relay network and one or more tracks, and the output consists of the optimal topologies for each track. In the following sections, we first present the solution for a single-track scenario, in which there is only one track in the system, and in Section 3.1.2, we generalize this approach to multiple tracks.

The relay network is given as a directed graph $G(V, E)$, with $v \in V$ being the relays and $e \in E$ the links between the relays. To describe the properties of a topology, we use a cost function $c : E \rightarrow \mathbb{N}$ and a delay function $d : E \rightarrow \mathbb{N}$ assigned to each link (or edge) of the relay network a cost indicated by $c_{ij} : (i, j) \in E$ and a delay value (also called latency) denoted by $d_{ij} : (i, j) \in E$. Here, the cost is an abstract value that may or may not directly correspond to an actual currency. Its main purpose is to differentiate links based on their *sensitivity* for traffic, i.e., whether or not we should avoid sending traffic through it. Delay values, on the other hand, are more realistically measurable quantities, even if they are only approximates of the exact delay on the path between the two relays it connects (e.g., when only considering propagation delay).

A track t is given with three attributes. Its publisher denoted by $P \in V$, its set of subscribers $s \in S \subseteq V$, and a delay budget (or delay bound) $D \in \mathbb{N}$ assigned to it, that is, the maximum tolerated E2E delay between the publisher and each subscriber. The E2E delay is measured independently for each subscriber by adding the delay values along the path between the publisher and the subscriber in the topology.

The roles of the publisher and the subscriber in a track are formalized by streams (or flows) indicated by $f \in F$, each with a path from the publisher of a subscriber across G , and with a vector R_f with a relay size in the stream path, which encodes roles of

the participating relays. Therefore, for each $f \in F$ and $i \in V$, R_i^f is described with the equation below.

$$R_i^f = \begin{cases} -1, & \text{if } i \in P \\ +1, & \text{if } i \in S \\ 0, & \text{otherwise} \end{cases}$$

The objective function can be characterized as the sum of link costs in the topology.

$$\sum_{e \in E_t} c(e)$$

Finally, the output of the optimization (i.e., the solution) is an optimal topology $G_t(V_t, E_t) : V_t \subseteq V, E_t \subseteq E$.

To measure link utilization, we introduce two other variables: $x_{ij}^f : (i, j) \in E, f \in F$ which represents the transmission bitrate of stream f on link (i, j) ; and $y_{ij} : (i, j) \in E$ that represents the link usage of the track on (i, j) .

There are two requirements for a topology to be optimal. First, the operation of the resulting relay topology must be cost-optimal and should minimize extra network usage. Second, it should enforce E2E latencies to meet QoS requirements.

To satisfy the first requirement (in this case, we do not have delay constraints in the model), the resulting Linear Program is formalized like this:

$$\begin{aligned} \min \quad & \sum_{(i,j) \in E} c_{ij} y_{ij} \\ & y_{ij} \geq x_{ij}^f \quad \forall (i, j) \in E, \forall f \in F \\ & \sum_{j:(j,i) \in E} x_{ji}^f - \sum_{j:(i,j) \in E} x_{ij}^f = R_i^f \quad \forall i \in V, \forall f \in F \\ & x_{ij}^f \geq 0 \quad \forall (i, j) \in E, \forall f \in F \\ & y_{ij} \geq 0 \quad \forall (i, j) \in E \end{aligned}$$

To include the delay bound, we first have to introduce an additional variable $z_{ij}^f \in \{0, 1\}$ to indicate whether the link (i, j) is on the path of stream f for every $f \in F$.

With this modification, the Linear Program becomes an Integer Linear Program formalized like this:

$$\begin{aligned}
\min \quad & \sum_{(i,j) \in E} c_{ij} y_{ij} \\
& y_{ij} \geq x_{ij}^f & \forall f \in F \\
& \sum_{j:(j,i) \in E} x_{ji}^f - \sum_{j:(i,j) \in E} x_{ij}^f = R_i^f & \forall i \in V, \forall f \in F \\
& x_{ij}^f \geq 0 & \forall (i,j) \in E, \forall f \in F \\
& y_{ij} \geq 0 & \forall (i,j) \in E \\
& z_{ij}^f * M \geq x_{ij}^f & \forall (i,j) \in E, \forall f \in F \\
& \sum_{i,j \in E} z_{ij}^f * d_{ij} \leq D & \forall f \in F
\end{aligned}$$

3.1.2 Problem Formulation for multiple tracks

In this section, we generalize the problem to multiple tracks. This means that we now allow for more than one track to be present in the system and, therefore, the topology must be optimized for each track individually. Following the pattern set in Section 3.1.1, we will begin by generalizing the relaxed problem without delay bound first, then move on to the variant with delay bound.

Let the set of tracks be $t \in T$. In this case, $P^t \in V$ and $s \in S^t \subseteq V$ denote the publishers and subscribers for each track $t \in T$, respectively. Similarly, $D^t \in \mathbb{N}$ denotes the delay bound, and $f \in F^t$ indicates the streams (or flows) for each track $t \in T$, and the value of R_i^f encoding node i 's role in flow f of a track t , for each $i \in V$ is described with the equation below.

$$R_i^f = \begin{cases} -1, & \text{if } i \in P^t \\ +1, & \text{if } i \in S^t \\ 0, & \text{otherwise} \end{cases}$$

In accordance to this, the objective function, presented for the single track problem, is summed for each track $t \in T$:

$$\sum_{t \in T} \sum_{e \in E_t} c(e)$$

Consequently, the solution to the problem is to assign a track topology G_t to each track $t \in T$.

Likewise, the variables in the Linear Program also change accordingly. The two, representing the transmission bitrate of stream f of track t on link (i, j) , and the link usage of track t on (i, j) become $x_{ij}^f : (i, j) \in E, f \in F^t$ and $y_{ij}^t : (i, j) \in E$ for each $t \in T$.

Correspondingly, the Linear Program is formalized as follows:

$$\begin{aligned}
\min \quad & \sum_{t \in T} \sum_{(i,j) \in E} c_{ij} y_{ij}^t \\
& y_{ij}^t \geq x_{ij}^f & \forall (i,j) \in E, \forall f \in F^t : \forall t \in T \\
& \sum_{j:(j,i) \in E} x_{ji}^f - \sum_{j:(i,j) \in E} x_{ij}^f = R_i^f & \forall i \in V, \forall f \in F^t : \forall t \in T \\
& x_{ij}^f \geq 0 & \forall (i,j) \in E, \forall f \in F^t : \forall t \in T \\
& y_{ij}^t \geq 0 & \forall (i,j) \in E, \forall t \in T
\end{aligned}$$

With the delay bounds included, the variable indicating whether the link (i, j) is on the path of stream f remains $z_{ij}^f \in \{0, 1\}$: link is on the path of stream f , with the addition of $f \in F^t : \forall t \in T$.

The resulting Integer Linear Program is formalized as follows:

$$\begin{aligned}
\min \quad & \sum_{t \in T} \sum_{(i,j) \in E} c_{ij} y_{ij}^t \\
& y_{ij}^t \geq x_{ij}^f & \forall f \in F^t : \forall t \in T \\
& \sum_{j:j,i \in E} x_{ji}^f - \sum_{j:i,j \in E} x_{ij}^f = R_i^f & \forall i \in V, \forall f \in F^t : \forall t \in T \\
& x_{ij}^f \geq 0 & \forall (i,j) \in E, \forall f \in F^t : \forall t \in T \\
& y_{ij}^t \geq 0 & \forall (i,j) \in E, \forall t \in T \\
& z_{ij}^f * M \geq x_{ij}^f & \forall (i,j) \in E, \forall f \in F^t : \forall t \in T \\
& \sum_{i,j \in E} z_{ij}^f * d_{ij} \leq D^f & \forall f \in F^t : \forall t \in T
\end{aligned}$$

3.2 Problem Classification

Previously, we formalized the problem at hand; however, as we have mentioned in Chapter 2, it seems to be very closely related to multicasting problems from a high-level perspective. It is thus worth discussing the findings from prior work in the field of multicasting to see what theoretical results they reached.

Based on the classification of multicast routing problems by Wang et al. [22], our problem is a tree-constrained tree optimization. Tree constrained, because it bounds on an aggregated metric (in this case, the E2E delay) along the path between each pair of terminal nodes, and tree optimization, because the objective function (in this case, minimization of total cost of tree's edges) is measured on the whole tree. This is also known as the Constrained Steiner Tree (CST) problem. Unfortunately, this combination is also an NP-complete problem. In their article, they also refer to known heuristics for the CST problem. Two notable ones are Zhu's Bounded Shortest Multicast Algorithm (BSMA) [23], and the KPP heuristic (Kompella, Pasquale, Polyzos) constructed by Kompella et al. [24]. Their

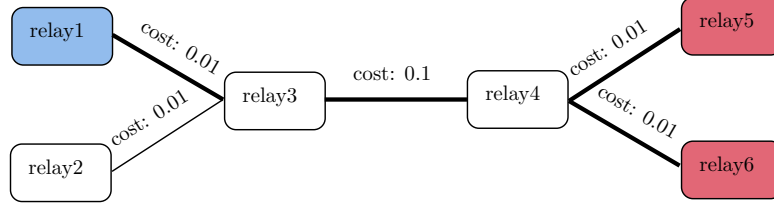


Figure 3.1: The *small_topo* topology satisfying the triangle inequality both in cost and in latency

algorithms share common characteristics; for example, both start by finding a suboptimal tree and iteratively improving on it by replacing edges with cheaper ones, while keeping the delay constraints satisfied.

However, our case is special because we are working at Layer 7 of the OSI model, which allows us to see a direct connection between every node. Therefore, our base network forms a full mesh. In this case, the Steiner Tree problem can reduce to the Minimum Spanning Tree (MST) problem under certain conditions, as can the constrained versions of the problems. This is promising because, while the Steiner Tree problem is NP-hard [25], the MST problem can be solved in polynomial time using either Prim’s or Kruskal’s algorithm.

The conditions presented below are sufficient conditions for this relaxation to work. With respect to both costs and latencies, the network must satisfy the triangle inequality. In the context of graphs, the definition of triangle inequality according to Black [26] is shown in Definition 1.

Definition 1 (Triangle inequality). The property that a complete weighted graph satisfies for all vertices u, v, w . Informally, the graph has no short cuts.

$$weight(u, v) \leq weight(u, w) + weight(w, v) \quad .$$

In Figure 3.1, we can see our example network, *small_topo*. We can conclude that it adheres to the triangle inequality, and therefore it is suitable for the relaxations described in the following. Note that we deliberately left out edges that would complete the network to a full mesh (otherwise known as a complete graph) in order to improve the understandability. However, it is easy to see that if the triangle inequality is true for this network, then the full mesh constructed by adding edges with summarized cost and latency values on the shortest paths between each two nodes, the resulting graph still satisfies the triangle inequality.

When these inequalities hold, the problem reduces to the Constrained Minimum Spanning Tree (CMST). Unfortunately, CMST is also NP-hard, although it is proven to be only weakly NP-complete [27]. Nevertheless, it is a useful property of our problem because it allows us to reduce the search space by filtering out nodes that are neither publishers nor subscribers.

In the following section, we try to find a close-to-optimal solution to the relaxed CMST problem with a heuristic algorithm that has polynomial complexity making it practical for real-world usage, where long processing times would make the ILP that could give optimal solution infeasible.

3.3 Heuristics

The problem with the integer linear programming-based solution lies in its complexity. Not only is it NP-complete [28], even for a considerably small network with only 5 to 10 active relays (i.e., relays with at least one publisher or subscriber connected to it), its runtime can easily take up minutes, therefore, it will not scale well for realistic scenarios.

Due to the long processing times of the linear programs, we present a simple heuristic approximation to find close-to-optimal routes for media streams over the relay networks. There were several decisions that we had to make when designing the algorithm. One notable was whether to make it dynamic or static in a sense that adding a new subscriber to be part of the distribution network does not require the re-computation of the existing topology. We have decided to make the algorithm dynamic, so that it is more easily applicable to real-world scenarios, as well as giving us possible increases in computational speed. However, this was much less of a concern for our test scenarios, since those are mostly based on static demands (with fixed subscribers). Also, since the complexity of the ILP was a key motivating factor in developing the heuristic solver, we employed a greedy strategy that may not always find the optimal solution but offers very minimal computational overhead.

The full mesh nature of the input graph allowed us to leverage the CMST relaxation. This enables to efficiently compute (locally) optimal connections to the distribution tree for new subscribers.

The intuition behind the algorithm is simple and can generally be described with two main ideas:

1. For each subscriber, we try to find a link that connects the subscriber to the existing distribution tree in a way that satisfies the delay constraints but is the least expensive among such links. If such a link exists, use it to create the new subscriber to the tree; otherwise, the problem is infeasible.
2. See if there is a way to reduce costs by reaching another subscriber with a link that goes through this newly added subscriber. If there is, choose the one with the lowest cost and the least E2E delay.

Notice how this algorithm can be easily converted to work with dynamically added subscribers. If the caller replaces the call to **multicast_heuristic** (Alg. 1) with a call straight to **add_subscriber** (Alg. 2) and consequently replaces the arguments passed, then it can add subscribers dynamically without needing to recompute the whole distribution tree.

For the sake of simplicity, we wrote this algorithm using global values to store common values used during the various optimization steps.

Algorithm 1 Heuristic inspired by Multicast Routing

Global variables: $G(V, E)$, t , ds , c , G_{tree}

procedure MULTICAST_HEURISTIC

$ds \leftarrow \{P^t: 0\}$

$c \leftarrow 0$

$tree \leftarrow \text{NEW_DIRECTED_GRAPH}(P^t)$

for $s \in S^t$ **do**

if **ADD_SUBSCRIBER**(s) **is null then**

return Infeasible.

return (c , $E_{G_{tree}}$)

Algorithm 2 Adding a subscriber

Global variables: $G(V, E)$, ds , c , G_{tree}

procedure ADD_SUBSCRIBER(v)

$e_{best} \leftarrow \text{MINIMAL_EDGE_FROM_NODE_TO_TREE}(G, v, G_{tree})$

if e_{best} **is null** **then**

return null

$v_{conn} \leftarrow e_{best}[0]$

$E_{G_{tree}} \leftarrow E_{G_{tree}} \cup \{e_{best}\}$

$c \leftarrow c + c_{e_{best}}$

$ds[v] \leftarrow ds[v_{conn}] + d_{e_{best}}$

AUGMENT(v)

return v_{conn}

Algorithm 3 Augmentation

Global variables: $G(V, E)$, t , ds , c , G_{tree}

procedure AUGMENT(v)

$repl_{best} \leftarrow (\text{null}, \text{null}, [], 0, \infty)$

$V_{loop} \leftarrow \text{FIND_NODES_TO}(v, G_{tree})$

for $v_{tree} \in V_{G_{tree}} \setminus V_{loop}$ **do**

$v_{prev} \leftarrow \text{GET_NODE_BEFORE}(v_{conn}, G_{tree})$

$e_{tbr} \leftarrow (v_{prev}, v_{tree})$

$e_{repl} \leftarrow (v, v_{tree})$

$d_{bal} \leftarrow (ds[v] + d_{e_{repl}}) - (ds[v_{prev}] + d_{e_{tbr}})$

$c_{bal} \leftarrow c_{e_{repl}} - c_{e_{tbr}}$

$G_{subtr} \leftarrow \text{SUBTREE_FROM_NODE}(v_{tree}, G_{tree})$

if ALL_SATISFY_DELAY($V_{G_{subtr}}$, d_{bal} , D^t) **and** $c_{bal} < repl_{best}.c_{bal}$ **then**

$repl_{best} \leftarrow (e_{repl}, e_{tbr}, G_{subtr}, d_{bal}, c_{bal})$

if $repl_{best}.c_{bal} < 0$ **or** ($repl_{best}.c_{bal} = 0$ **and** $repl_{best}.d_{bal} < 0$) **then**

$E_{G_{tree}} \leftarrow E_{G_{tree}} \cup \{repl_{best}.e_{repl}\} \setminus \{repl_{best}.e_{tbr}\}$

$c \leftarrow c + repl_{best}.c_{bal}$

for $v_{subtr} \in V_{G_{subtr}}$ **do**

$ds[v_{subtr}] \leftarrow ds[v_{subtr}] + repl_{best}.d_{bal}$

3.3.1 Complexity Analysis

We describe the complexity of the algorithm in a top-down approach.

However, it is important to note that some assumptions have to be made regarding the underlying graph's representation and the related functions. Namely, we assume that the lookup time of an edge's cost is $O(1)$; finding the reverse edges of a node is $O(1)$; inserting a node into the tree is $O(1)$; and set and map lookup times are $O(1)$. These assumptions can be made without loss of generality, as they are typically satisfied when using efficient data structures and representing graphs with adjacency lists (using hash maps) instead of adjacency matrices.

The **multicast_heuristic** function first initializes the common variables in $O(1)$, then it applies **add_subscriber** to each and every node in the list of subscribers adding an $O(n)$ factor to its complexity.

The **add_subscriber** function first tries to find a suitable edge by checking each node in the tree which is done in $O(n)$. After that, it updates the common values in $O(1)$. Finally, it invokes **augment**.

The **augment** (Alg. 3) function first determines the nodes in the tree that would cause loops if they were connected to *node*. Specifically, these nodes are *node*'s predecessors in the tree (including *node* itself). Calculating the predecessors in the tree up until the root node (P^t) takes $O(n)$. After determining V_{loop} , each v_{tree} (that is not loop causing) is checked individually, which adds an $O(n)$ factor to its complexity. During each iteration, a G_{subtr} of G_{tree} is discovered through a search, after which each node of G_{subtr} is checked if its latency requirements would be satisfied if reached from *node* instead of their current connection to the *tree*. The search can happen with depth-first search (DFS) or breadth-first search (BFS) in $O(m + n)$, but since the properties of trees always hold on *tree*, m is at most $n - 1$, thus the complexity of the operation reduces to $O(n)$. The latency check can also be done in $O(n)$. After every augmentation possibility was checked, the best one (in terms of cost and latency reduction) gets selected, and the augmented latency values of G_{subtr} 's nodes have to be reduced in $O(n)$.

The complexity of each function is summarized below.

- **augment:** $O(n) + O(n) \cdot [(O(n) + O(n))] + O(n) \approx O(n) + O(n^2) + O(n) \approx O(n^2)$
- **add_subscriber:** $O(n) + O(n^2) \approx O(n^2)$
- **multicast_heuristic:** $O(n) \cdot O(n^2) \approx O(n^3)$

This concludes that the total complexity of the presented heuristic algorithm is $O(n^3)$.

3.4 Summary

In this Section, we first formulated the media routing problem, then we gave an optimal solution in the form of an integer linear program, and concluded that due to its complexity, it is not expected to scale well in practice. Next, we have discussed known heuristics for similar problems, then we have shown an applicable relaxation to the problem, and we have provided a novel algorithm to find a close-to-optimal solution to our relaxed problem while having a complexity of $O(n^3)$, which makes it suitable for use in real life.

Chapter 4

Implementation

Based on our theoretical findings, we extended MoQ with media routing capabilities that make media distribution cost-efficient and latency-aware. In addition, we have created a numerical evaluation tool to validate our algorithms. To confirm our theoretical results and assess the feasibility of using our model as a media routing extension to MoQ, we devised two interdependent solutions. Firstly, we created an optimizer prototype that implements the integer linear program and the multicast heuristic algorithm described in Sections 3.1.1 and 3.3. Secondly, we enhanced moq-rs to enable support for dynamic topology construction. In the subsequent sections, we will initially outline the high-level architecture along with the primary design decisions considered, followed by a gradual in-depth analysis. The source code is publicly available on GitHub for the Optimizer and our moq-rs fork [29].

4.1 Architecture

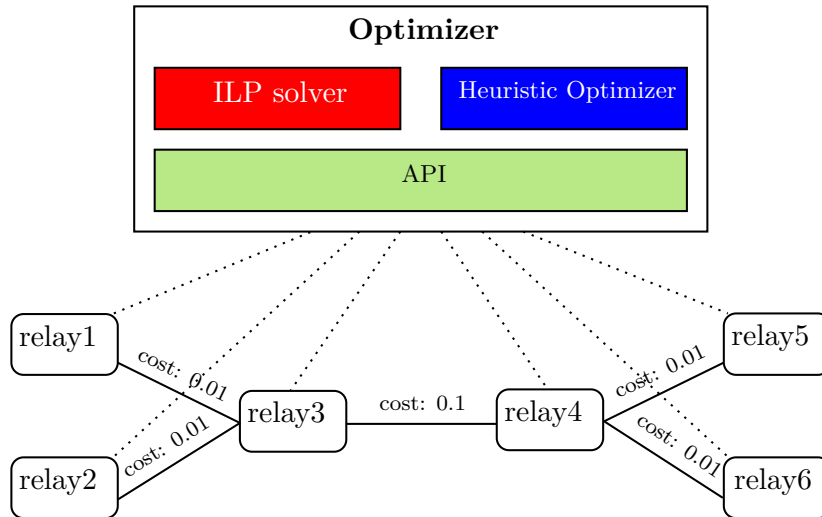


Figure 4.1: The Optimizer and the relays

We strove to make all control in our system centralized, so that relays are fully coordinated. In Figure 4.1, we can see how the Optimizer acts as a single source of truth controlling every relay in the system, resembling the widely-used Software-Defined Networking architecture [30].

4.2 Optimizer

The Optimizer component consists of two subcomponents: the topology optimizer and the API that provides access to the former through endpoints. Initially, these were separate applications that communicated with one another; however, it eventually became more efficient to consolidate them into a single one.

API For simplicity reasons, we designed the API to work in the following manner. Every time a new subscriber joins, the moq-relay will invoke the corresponding endpoint on the optimizer API, instead of using the original moq-api, which will immediately start to calculate the new optimal topology after receiving the request. Consequently, this means that the optimizer API is synchronous, despite the fact that in real systems asynchronous APIs are usually preferred due to performance considerations. However, the relays should only really invoke the optimizer’s endpoints when subscriptions or tracks change; thus, it would only affect the Time To First Byte (TTFB) metric at subscribers, which is beyond the scope of our work. For the implementation of the API subcomponent, we have used FastAPI due to its ease of use [31].

Topology Optimizer The optimizer determines an optimal, or near-optimal, topology for distributing a specified track throughout the network. It takes as input a network containing all system relays, along with the track for which the topology needs to be optimized. The optimizer employs multiple strategies, including the state-of-the-art Direct Link Tree solution, an Integer Linear Program solver, a Multicast Heuristic, and a Minimum Spanning Tree-based approach. To support these strategies, we use NetworkX, a widely adopted Python library for graph manipulation, providing the data structures needed to represent graphs in our code [32].

4.3 Testbed

We aimed to create various small yet realistic topologies to conduct real-world tests following accurate simulations. All components, except for the publisher and subscriber clients, were deployed within Mininet, along with the required number of relays. All relays are connected to the chosen API, either the moq-api or the optimizer, which makes the same endpoints available as the original moq-api. Mininet [30], a network simulation framework for Python, allows for the simple construction of virtual networks and the execution of client hosts within these networks. Ultimately, the general topology appeared as follows: Each relay was assigned a unique IP address on its loopback interface and was directly linked to all the others. We constructed the desired topology by imposing their specified delays on the connections between each pair. Each relay had an interface featuring an IP address from a distinct network, connected to the switch adjacent to the API. Furthermore, by configuring the appropriate `ip route` on the neighboring relays, which effectively functioned as isolated Linux hosts, these relays were able to acquire routes to the IP addresses on their neighbors’ loopback interfaces. This setup bypassed the additional complexity typically introduced by various routing protocols. The publishing and subscribing clients were then connected to their assigned relay via a single interface.

In order to depict the described topology and achieve precise measurements, we designated several interfaces for each relay host. Initially, every relay was linked to others via their underlay connections if there was one in the topology description, with provided latencies.

Following this, an overlay link was created for each relay that lacked a direct underlay connection to another relay, with a latency equal to the total latency of the associated sequence of underlay links. Finally, an extra interface was linked to a virtual switch, which was also connected to the API. These interfaces utilized a distinct subnet from the previous configurations, enabling straightforward filtering of API communication while summarizing interface counters.

We determined the necessity to measure latency and interface usage. For the latter, we aggregated the transmitted bytes from each relay network interface after executing each scenario. Although this description simplifies the process, more detail will be provided in Figure 5.2.

For straightforward delay measurements, one attempt could be to adapt the existing `moq-rs` crate, `moq-clock` which can send the publishers current time and date as data on a MoQ track. However, we found that to be inappropriate, as subtracting timestamps led to extremely high initial latencies. As noted in Section 2.1, the creator of `moq-rs` is also involved in a GStreamer component. Another possibility for measuring delay was to use the `timestampoverlay` GStreamer element [33]. This intermediary component can append a sort of barcode to each video frame, which upon reception can be decoded to calculate the time taken to reach it. Coupled with `moq-gst`, it enables the assessment of the average latency for a video transmitted over the network. However, encoding after barcode addition and decoding on the receiver side could introduce an extra, inconsistent delay. Therefore, before each scenario, we need to perform a baseline measurement with just one relay, a subscriber, and a publisher, connecting to the relay to measure latency on this minimal route. This allows us to subtract this baseline value from actual network measurements, isolating latency without media conversion delays. For verification, we also print the deviation of the measured latency to identify any hardware anomalies.

To exactly highlight the effectiveness of the media routing capabilities of the optimizer, we could not use this. Somehow we had to add the capability to the API to handle the different routes it should use. For this first, we modified the original `moq-rs`'s `moq-api` and `moq-relay` crate, but later added these few HTTP endpoints to the Optimizer's API. Although originally having GET, POST and DELETE endpoints at `/origin/<track name>`, later we have decided to change that to `/origin/<relay>/<track name>` in order to include a relay identifier after the origin in the case of POST and GET.

When a publisher connects to a relay to send a stream, the relay will tell the API that information. The API will know the entire topology and - by the time - will also know the publishing relay. After that, the optimizing API will think of an optimal next hop for each other relay. These next hops will be available by searching for the specific track and the relay's id. If there are multiple hops each time a next hop relay is contacted, it will also search for that track as a local track, and after not finding it it will ask the API about where should it ask for it. So, the API will respond with the next hop relay. And this will continue until we reach the publisher's relay.

Chapter 5

Evaluation

After introducing the technologies used and thoroughly describing our mathematical model, we evaluated our methods for scalability and practical usability using a computer with an AMD EPYC 7502P CPU (32 cores) and 128 GB RAM, running Ubuntu 22.04 with Python 3.10 and Rust 1.77.2 installed. We adjusted the settings by switching to performance mode and fine-tuning network configuration parameters.

Our evaluation goals are dual: We want to compare the theoretical complexities of the different solvers presented to their empirical performance. However, we also want to demonstrate the usability of our system with realistic components. In Section 5.1, we test our theoretical solutions on a benchmark to see how they scale in a simulated environment, then in Section 5.2 we use the testbed described in Section 4.3 to see how the system behaves when using realistic components.

5.1 Benchmark

First, we evaluate the performance of the media routing algorithms using numerical evaluation. For this reason, we have constructed another synthetic network that has many times more relays than *small_topo*, making it suitable for testing at scale. We aimed to create a realistic network model by placing relays in various Microsoft Azure Regions, using approximate geographical locations based on publicly available information [34]. Furthermore, we incorporated data inspired by the L3 topology of the GÉANT research network [35] to simulate the underlay of a large-scale network. However, we only used this underlay for the cost calculation and took a different strategy for the delay calculation.

The GÉANT map we used includes cities that each connected to one or more other cities through links. To help mapping from underlay to overlay, we connected each relay in the overlay to its nearest city in the underlay. For these “last-mile” connections, we did not assign costs. Cities are grouped into regions on the GÉANT map, and we could leverage this in our cost assignments for the underlay links. The interregional links were assigned a cost equal to the number of cities, while the intraregional links received a cost of 1. Finally, we established a full mesh network by connecting each unique pair of relays with a link. The delay values for each of these overlay links were derived from the straight-line distance between them and the cost was calculated using the described method.

Putting together the names of these data sources inspired us to name it the *Azure-GÉANT* network, and from now on we will refer to this network like that. After assembling it as

the network, we have measured it and concluded that it satisfies the triangle inequality in terms of both latency and cost described in Section 3.2, which confirms the intuitions.

To compare the complexity of the different optimizers at scale, we performed simulations on *Azure-GÉANT* and measured the total time taken to optimize the topology, as well as the parameters of the optimized topology (cost and average E2E latency). The optimizers included in the benchmarks were: (1) Direct Link Tree; (2) Multicast Heuristic; (3) Integer Linear Program; and (4) Minimum Spanning Tree. Optimizers (2) and (3) have been well described in Chapter 3, (1) and (4), on the other hand, are implementations for the two extreme points of the delay-cost spectrum introduced in Chapter 1 which we have included for comparison.

We have set up the simulation with a total of 24 peers (geographically located in multiple continents), of which an ever-increasing subset is selected for the track (with the first peer included in the subset being the publisher and every other a subscriber) in each test run of the simulation. One thing to note is that we have not run simulations for less than two peers, because in that case no media routing would take place. In each simulation run, we have measured two metrics: the total cost and the maximum E2E latency present in the topology.

In Figure 5.1, we can see the results as the number of peers increases. Note that for better visualization of orders of magnitude, we have plotted the runtime values on a logarithmic scale. Note when the cost for a specific optimizer’s result is zero, it means that the optimizer could not produce a topology that would meet the delay bound. We can see this in action in the case of $D = 75$ ms. In this case, after 4 peers are in the track, MST cannot satisfy the delay bound, and thus yields back a cost of zero.

We also get confirmations on the theoretical complexity of each algorithm. ILP follows an exponential curve due to its NP-completeness, and by analyzing the algorithmic complexity of the Multicast Heuristic, we concluded that it is in $O(n^3)$.

Direct Link Tree is an outlier here, as it is not an actual media routing method and is only included for comparison reasons. This makes reasoning about its complexity difficult, as its current implementation (in the form of no media routing) is a distributed solution where there is no optimizer involved in the system; thus, when relays need to reach for a given track, they invoke an endpoint of `moq-api`, which in turn gives back the next-hop relay in $O(1)$. However, when simulating the effect of no media routing, we have constructed the Direct Link Tree optimizer to be compatible with the internal interface between the different kinds of optimizers and the API. Therefore, the algorithm that generates direct links between the publisher and subscriber relays has a complexity of $O(n)$.

Our MST-based solver uses the NetworkX library, which provides 3 choices for the MST implementation: Kruskal, Prim, or Borůvka. But regardless of which is chosen, the complexity is guaranteed to be below $O(m \cdot \log n) \approx O(n^2 \cdot \log n)$.

We can see this in practice, as the multicast heuristic takes more time to compute than the MST after around 12 peers. The figure shows that the idea of the delay-cost spectrum does not only exist in theory but is present in practical scenarios as well. The Direct Link Tree gives a lower bound on the achievable latency at the expense of higher operational cost, at which it controversially provides an upper bound. On the contrary, while the MST gives a lower bound on operational cost, it does not necessarily give an upper bound on the latency, and even when it does, it does not fall behind as much from the other optimizer types’ latency values. In fact, MST and ILP behave very similarly while the delay bound is satisfied, and only after that will the ILP strongly deviate from the MST. That is expected, as we have previously concluded in Section 3.2, as in networks where

the triangle inequality holds, the problem reduces to a variant of MST, CMST. Therefore, as long as the constraints are satisfied, the two should yield similar results.

A pattern resembling a stair step pattern is observable in the histogram of all optimizers. This is likely due to the drastic difference in cost and delay of links spanning over regions (sometimes even continents) compared to links connecting relays in the same region.

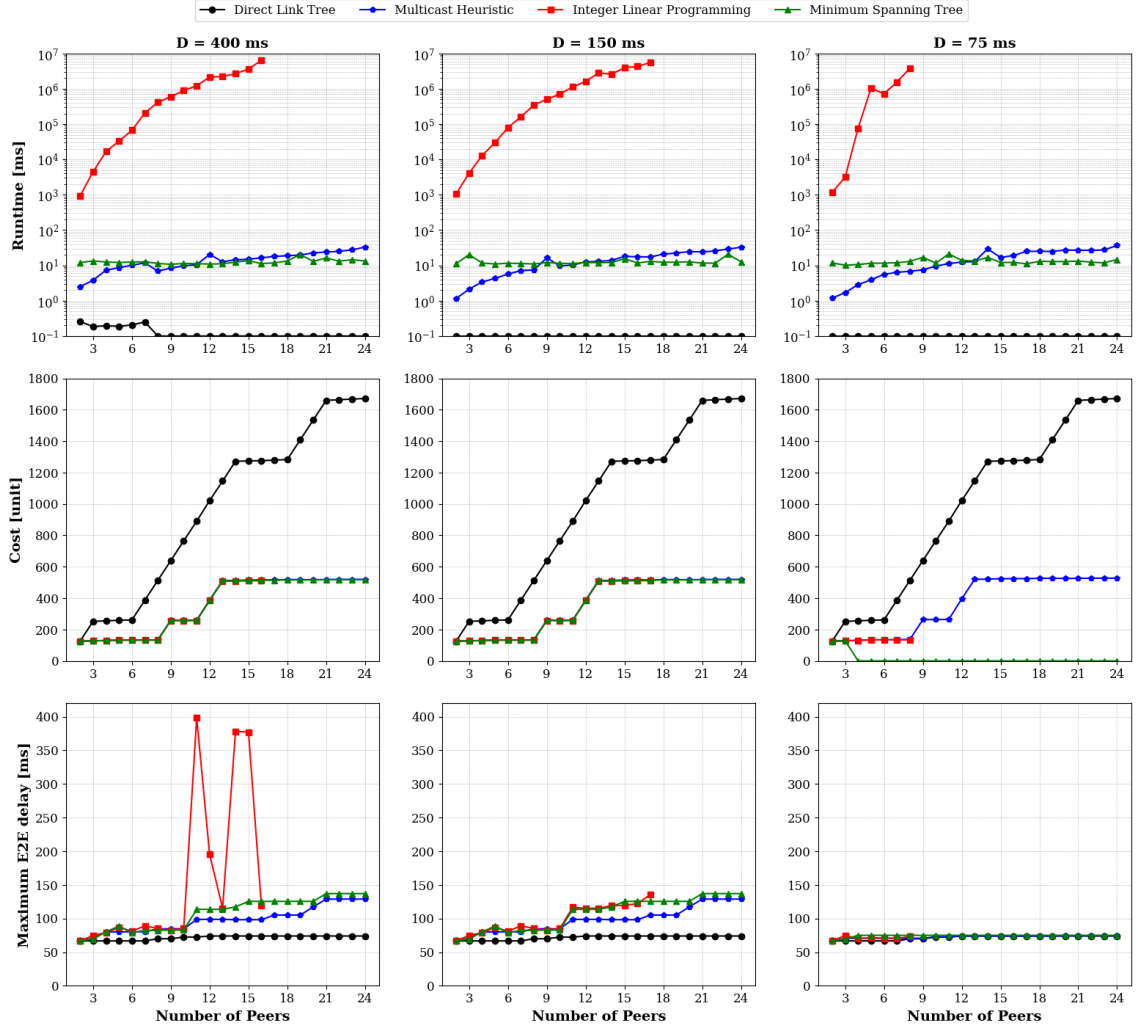


Figure 5.1: Complexity of Optimizers for Content Types and Number of Peers

5.2 Testbed

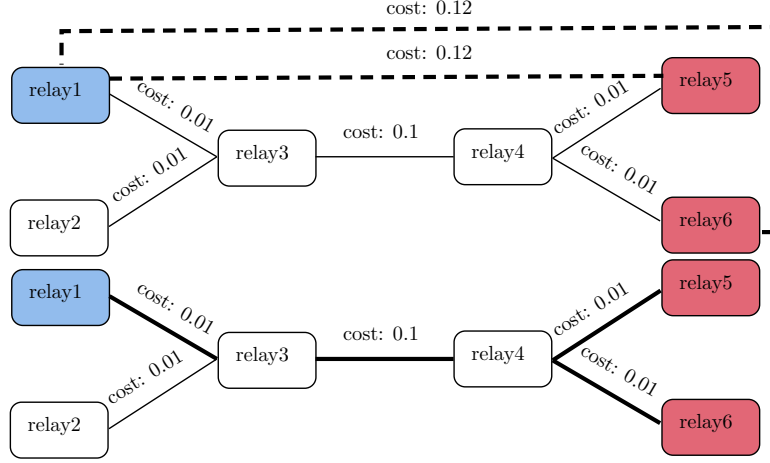


Figure 5.2: Links in using the original (top) and the optimized API (bottom), the chosen routes displayed with highlighted lines

In this section, we will use the testbed discussed previously, which operates in Mininet, to evaluate the optimizer and its algorithms in a realistic environment. Our implementation can be easily started by providing the topology description file and selecting the desired API. We can opt for the Original API, which consistently returns an overlay link, theoretically establishing a connection through IP routing. Alternatively, we can choose the Optimizer API, which always provides the next hop relay up to the publisher’s relay, creating a route with more hops but reducing link usage for multiple subscribers. It is worth mentioning that for simplicity, we only used the optimal integer linear programming-based optimizer through the Optimizer API in our measurements.

In Figure 5.2 we can see the used links in the often-mentioned *small_topo* for the two subscribers with the highlighted links. The other overlay links, the exact clients at relay1, 5, and 6 are not displayed to keep the figure clearer as before. But actual API calls are done in the same way as in Figure 2.3 at every relay in the Optimized case and the relay5 and 6 in the Original case. In the former, one can see that the total cost will be 0.24, using the two overlay links. Although we only use two links in this scenario, later in the testbed’s evaluating phase we will take into account the 2 times 3 underlay links that the traffic would use in a real-life scenario. It is important to note that in this case the data has to be duplicated at the start, which is why we have two “dashed routes” coming from relay1. In the other case the cost will be 0.13 using the underlay links, and only splitting at relay4. These costs were consistent throughout our measurements.

After concluding that the central optimized API could effectively choose a cost-efficient route, we should review our other measured values as presented in Table 5.1. Both figures illustrate the averages of all our measurements, though their minimum and maximum values do not deviate significantly from these averages. We measured both scenarios 10 times. The Original and Optimized APIs show a variance of approximately ± 0.22 or less from the average. Our testbed allowed us to obtain the latency for each subscriber; we are displaying the average latencies of each measurement. When using the optimized API, it was observed that the second subscriber experiences a few milliseconds additional delay compared to the first, a discrepancy not noted with the Original API. A takeaway can be

that the Optimized configuration is a bit slower than the Original. This was anticipated, as in the Original scenario, the subscriber’s relay only needs to query the central API once to locate the track, followed by connecting to a single relay in Layer 7, the publisher’s relay, to manage the transmission. In contrast, in the Optimized scenario illustrated at the bottom of Figure 5.2, two subscriber relays, along with two others, query the central API and manage the transmission. Thus, we can conclude that the processing latency introduced by the relays is measurable but is not significant.

The more apparent benefit of the Optimized API becomes clear when examining the bits sent over the network. By using only four underlay links instead of the two overlay links (which would be translated to 6 underlay links), we got around 13 MB of data instead of 40 MB. The deviation here is below ± 0.05 MB for the Optimized case and a bit higher ± 0.14 and ± 0.17 MB for the Original scenario.

Measured feature	API	Average	Minimum	Maximum
Average latency at subscribers (ms)	Original	45.65	45.501	45.852
	Optimized	48.94	48.716	49.117
Transmitted megabytes	Original	40.734	40.56	40.876
	Optimized	13.56	13.523	13.6

Table 5.1: Measured latency and network usage

To conclude our evaluation, the benchmark results show a clear delay-cost trade-off between optimizers, which confirms practical evidence for the theoretical delay-cost spectrum. The Direct Link Tree minimized latency at a higher operational cost, while MST reduced costs but did not necessarily optimize latency. Testing with our Mininet-based testbed further confirmed the cost-efficiency of an optimized API routing structure, yielding significantly lower network load but with a slight increase in average end-to-end latency introduced by the relays handling the traffic at the application layer. Thus, this chapter demonstrates that while optimized media routing can reduce link usage effectively, they introduce minimal delays, making them a practical choice for scalable media routing for MoQ.

Chapter 6

Conclusion

In summary, we have addressed the challenge of real-time content distribution over public networks by developing an optimized relay topology approach inspired by CDN and multicast solutions. The majority of global Internet traffic involves media and is sensitive to latency, with WebRTC serving as the prevailing solution. Deploying and scaling WebRTC in cloud environments is challenging due to its complexity. To address this issue, the IETF introduced Media over QUIC. Upon the introduction of this new media protocol, we addressed the lack of media routing within its current implementation. We began by examining large-scale media transmission methods, and identified limitations that made them unsuitable for our goals. We then presented a comprehensive approach to modeling and optimizing relay topologies, including formulations for single-track and multi-track scenarios, along with advanced techniques to reduce problem complexity. To achieve practical usability, we have introduced a heuristic algorithm that provides close-to-optimal solutions in polynomial time. Finally, we have put our solutions under benchmarks and tested it on a testbed we have constructed to see the realistic results. Our evaluation on sample networks demonstrated the scalability and efficiency of the solutions, highlighting improvements in cost and delay performance compared to the state-of-the-art using no media routing.

6.1 Future Work

This project can be expanded in numerous directions. Future work could focus on the extension of the media routing model toward capacity awareness. In this case, the model would also have to consider the actual bandwidth usage of each track on the links, which in total (i.e., summed for all tracks using that link) must not exceed the capacity of each link. Such a model would render the assumption about the independency of tracks obsolete.

In addition, one approach to analyzing the protocol involves checking network performance when multiple tracks are present, assuming that hardware limitations are not exceeded. At the time of writing, certain features were absent from the protocol, such as the ability to dynamically remove subscribers from the testbed. However, it is anticipated that the GOAWAY control message will later fulfill this requirement, potentially leading to new testbed applications. Testing subscriber migration will be one of these possibilities.

Finally, in this work, we have implicitly assumed that a publisher or a subscriber may not connect to more than one relay at a time; however, an interesting area for research could be multipath media routing.

Acknowledgment

Special thanks to Felicián Németh, István Pelle, and Gergely Pongrácz for the discussions that led to this line of work. We thank Ericsson Hungary for providing the infrastructure to run our measurements. We also express our gratitude to Luke Curley and the supportive moq-rs community.

Bibliography

- [1] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach (8th Edition)*. Pearson, 8th edition, 2020.
- [2] Sandvine. The global internet phenomena report 2023. Technical report, Sandvine, 2023.
- [3] Mark R Johnson. Behind the streams: The off-camera labour of game live streaming. *Games and Culture*, 16(8):1001–1020, 2021.
- [4] Twitch Support. Stream together. <https://help.twitch.tv/s/article/stream-together>, Accessed: 2024-10-31.
- [5] Phenix Real Time Solutions, Inc. Latency comparison of common streaming services for the 2024 super bowl. <https://www.phenixrts.com/resource/super-bowl-2024>, Accessed: 2024-10-31.
- [6] International Telecommunication Union. Itu-t recommendation g.114: One-way transmission time. ITU-T Recommendation G.114, International Telecommunication Union, Geneva, 2003.
- [7] Luke Curley. Replacing webrtc with quic. <https://quic.video/blog/replacing-webrtc/>, Accessed: 2024-10-31.
- [8] Luke Curley. Media over quic: Big buck bunny demo. <https://quic.video/watch/bbb>, Accessed: 2024-10-31.
- [9] Nestri. Nestri | play with friends on chrome. <https://nestri.io>, 2024. Accessed: 2024-10-05.
- [10] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the ip multicast service and architecture. *IEEE network*, 14(1):78–88, 2000.
- [11] Mojtaba Hosseini, Dewan Tanvir Ahmed, Shervin Shirmohammadi, and Nicolas D Georganas. A survey of application-layer multicast protocols. *IEEE Communications Surveys & Tutorials*, 9(3):58–74, 2007.
- [12] Adrian Popescu, Demetres D Kouvatsos, David Remondo, and Stefano Giordano. Content distribution over ip: developments and challenges. In *Network Performance Engineering: A Handbook on Convergent Multi-Service Networks and Next Generation Internet*, pages 979–987. Springer, 2011.
- [13] Netflix. Netflix open connect. <https://openconnect.netflix.com>, 2024. Accessed: 2024-09-14.
- [14] Dan Rayburn. Global cdn services market about \$5b in 2023, expected to grow 3% in 2024, driven by aws. <https://www.streamingmediablog.com/2024/07/cdn-market-size.html>, 2024. Accessed: 2024-10-31.
- [15] Larry Peterson and Bruce Davie. Computer networks: A systems approach - multicast. <https://book.systemsapproach.org/scaling/multicast.html>, 2024. Accessed: 2024-10-31.
- [16] Ashok Sharma. What is iptv? <https://www.engineersgarage.com/what-is-iptv/>, Accessed: 2024-10-31.

- [17] Ilyin. Ncdn - cdn, 2022.
- [18] Iyengar and Thomson. Quic: A udp-based multiplexed and secure transport. <https://datatracker.ietf.org/doc/html/rfc9000>, Accessed: 2024-10-31.
- [19] Will Law. *WebTransport Working Group Updates*. IETF, W3, 2021. <https://www.w3.org/2021/10/TPAC/demos/webtransport.html>.
- [20] M. Zanaty et al. *Low Overhead Media Container*, 2024. <https://datatracker.ietf.org/doc/html/draft-mzanaty-moq-loc-03>.
- [21] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Pearson, 6th edition, 2020.
- [22] Bin Wang and Jennifer C Hou. Multicast routing and its qos extension: problems, algorithms, and protocols. *IEEE network*, 14(1):22–36, 2000.
- [23] Qing Zhu, Mehrdad Parsa, and JJ Garcia-Luna-Aceves. A source-based algorithm for delay-constrained minimum-cost multicasting. In *Proceedings of INFOCOM’95*, volume 1, pages 377–385. IEEE, 1995.
- [24] Vachaspathi P Kompella, Joseph C Pasquale, and George C Polyzos. Multicast routing for multimedia communication. *IEEE/ACM transactions on Networking*, 1(3):286–292, 1993.
- [25] Viggo Kann. Approximation algorithms. <https://www.csc.kth.se/~viggo/wwwcompendium/node78.html>, n.d. Compendium of Algorithms and Data Structures.
- [26] Paul E. Black, editor. *Algorithms and Theory of Computation Handbook*. CRC Press LLC, 1999. *Dictionary of Algorithms and Data Structures [online]*.
- [27] Ram Ravi and Michel X Goemans. The constrained minimum spanning tree problem. In *Algorithm Theory—SWAT’96: 5th Scandinavian Workshop on Algorithm Theory Reykjavik, Iceland, July 3–5, 1996 Proceedings 5*, pages 66–75. Springer, 1996.
- [28] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 2010.
- [29] HSN Lab. Github repository of our cdn optimizer and testbed. <https://github.com/hsnlab/moq-relay-topo-opt>, Accessed: 2024-10-31.
- [30] Bob Lantz and the Mininet Contributors. Mininet overview. <https://mininet.org/overview/>, Accessed: 2024-10-31.
- [31] Sebastián Ramírez. Fastapi: Fastapi framework, high performance, easy to learn, fast to code, ready for production. <https://fastapi.tiangolo.com/>, 2024. Accessed: 2024-10-31.
- [32] NetworkX. Networkx: Network analysis in python. <https://networkx.org/>, Accessed: 2024-10-31.
- [33] Smart Networks and Services Joint. gst-timestamper. <https://github.com/SNS-JU/6gxr-latency-clock>, Accessed: 2024-10-31.
- [34] Microsoft. Explore the globe of microsoft data centers. <https://datacenters.microsoft.com/globe/explore>, Accessed: 2024-10-31.
- [35] GÉANT Association. GÉant connectivity map. <https://map.geant.org/>, Accessed: 2024-10-31.