Popularity-Based Prefetching for Video Streaming in Named Data Networking

by

Henrik Tambo Buhl

201905590

Master's Thesis

in

Computer Engineering

Supervisor: Rune Hylsberg Jacobsen

Aarhus University, Department of Electrical and Computer Engineering

2 January 2025

Abstract

Named Data Networking (NDN) offers a content-centric approach to networking, shifting the focus from host-based communication to efficient content retrieval. This project investigates a prefetching method designed to cache popular video segments within NDN, leveraging user engagement data to predict and store frequently accessed content closer to consumers.

The proposed prefetching mechanism was evaluated across multiple scenarios using metrics such as cache hit ratio, average delay, retransmissions, hop count, and path reduction. The results demonstrate that prefetching popular video segments significantly improves content availability and reduces retrieval delays. However, the effectiveness of the approach is enhanced when combined with robust replacement policies, such as FIFO and LRU, which maintain balanced caching performance across varying network conditions.

This study underscores the potential of prefetching mechanisms in enhancing video streaming within NDN. It highlights the importance of adaptive, network-aware strategies to optimize caching efficiency and scalability. Future work will focus on refining the prefetching module by integrating dynamic content prediction models and hybrid caching policies to further improve the performance of NDN-based content delivery systems.

Preface

This report concludes my Master's degree in Computer Engineering at Aarhus University. I would like to thank my academic supervisor Rune Hylsberg Jacobsen for making this project possible. I would also like to thank my bestfriend Alexander Stæhr Johansen for always being there for me when I need him and helping me through tough times.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

The following chapter starts by motivating the project, highlighting current challenges in video streaming and the limitations of traditional, host-centric architectures. After this, the purpose of the project is presented, accompanied by its associated goals, and a description of the approach taken to address these goals. Finally, the chapter concludes with an overview of the report's structure, material, and a brief mention of the code repository used in this work.

1.1 Motivation

The modern internet, largely built on the *TCP/IP architecture*, faces challenges in optimizing content delivery, particularly for video streaming. This host-centric model, which routes data based on server locations (IP addresses), struggles to accommodate the ever-increasing demand for high-quality, on-demand video content. As a result, inefficiencies such as high latency, bandwidth waste, and poor Quality of Experience (QoE) arise [1].

To overcome these limitations, alternative network architectures like *Named Data Networking (NDN)* have been proposed. Unlike the host-based paradigm, NDN shifts to **content-based networking**, where data is requested by name rather than by server location. This enables **in-network caching**, allowing frequently accessed content to be stored closer to the user [2]. NDN has shown promise in video streaming scenarios by improving data retrieval speeds and reducing network congestion [3]–[5].

However, challenges remain. Traditional prefetching strategies often assume that users watch videos sequentially from start to finish. In reality, especially on platforms like YouTube, user behavior is much more dynamic: viewers often skip sections or repeatedly watch particular segments. Without accounting for these patterns, sequential prefetching leads to wasted bandwidth and inefficient caching [6], [7].

1.2 Purpose

The purpose of this project is to explore how user engagement data specifically information about which segments of a video are frequently viewed or skipped can inform more intelligent prefetching strategies in NDN. By examining non-linear

viewing behaviors, the project aims to move beyond the assumptions of sequential consumption. Instead, it seeks to leverage aggregated user interaction patterns to guide network-level decisions about which content should be prefetched and cached closer to end-users.

1.3 Project Goal

The primary goal of this project is to design, implement, and evaluate a **popularity-based prefetching system** integrated with NDN. By analyzing user behavior patterns from platforms like YouTube, this system will:

- Identify segments of content that are frequently accessed or skipped.
- Prefetch and cache popular segments closer to the user to reduce latency.
- Minimize unnecessary data transfers and improve overall QoE.

In doing so, the project seeks to address the inefficiencies in current prefetching techniques and demonstrate how a data-driven approach can enhance video streaming performance in content-centric networks.

1.4 My Approach

The approach taken in this project involves several key steps:

- **Data Collection Analysis:** Aggregate user engagement data from large-scale video platforms (such as YouTube) to understand viewing patterns.
- **Popularity Profiling:** Determine video segment popularity based on frequently watched portions and skipped segments.
- **NDN Integration:** Integrate these popularity insights into an NDN-based prefetching mechanism, ensuring that in-demand segments are cached closer to end-users.
- **Evaluation:**Assess the system's performance through metrics such as Cache Hit Ratio, improved data retrieval times, the Hop Count, and amount of retransmission. These metrics should help in determining whether our module has enhanced the QoE.

1.5 Report Structure and Material

The remainder of this report is organized as follows:

- **Chapter 2**: Provides an overview of Named Data Networking (NDN) principles and compares them with TCP/IP, highlighting the challenges associated with both architectures. This chapter establishes the context for the research by addressing the limitations of traditional networking and the potential advantages of NDN.
- **Chapter 3**: Introduces the design concepts for the proposed prefetching mechanism, including the use of user engagement data to predict popular video segments and integrate them into the NDN caching framework.
- **Chapter 4**: Details the implementation of the proposed prefetching mechanism. It includes the data extraction pipeline for analyzing user behavior and discusses how these insights are integrated into the NDN architecture. Furthermore, it presents the simulation setup and the scenarios used for evaluation.
- **Chapter 5**: Presents the evaluation results, illustrating the performance differences between the various scenarios and highlighting the strengths and limitations of the proposed prefetching system. Metrics such as cache hit ratio, average delay, path reduction, retransmissions, and hop count are used to evaluate system performance comprehensively.
- **Chapter 6**: Discusses the implications of the results, highlighting potential areas for improvement in prefetching strategies. It also identifies limitations of the current approach and proposes future research directions.
- **Chapter 7**: Concludes the report by summarizing the key contributions, emphasizing the potential of popularity-based prefetching to improve video streaming performance in NDN, and outlining the significance of this work for future research.

1.6 Code Repository

The code and tools used to extract and process user engagement data are available in the following GitHub repository: GitHub.[1]

A more detailed description of the repository's functionality and how it supports the project will be provided in later chapters.

[1]Code repository: https://github.com/Henrikls/ytp_heatmap_data_extractor/tree/main

Chapter 2

Background

This chapter introduces the fundamentals of the project. The chapter gives an overview of the traditional TCP/IP based architecture, whereafter it dives into the fundamentals of NDN and thereafter video streaming and prefetching methods followed by popularity-based prefetching and at last related works.

2.1 TCP/IP-Based Architectures and Video Streaming

2.1.1 Introduction to TCP/IP

The **Transmission Control Protocol/Internet Protocol (TCP/IP)** suite has been the foundation of the internet since its inception. TCP/IP provides the essential mechanisms for packet-switched data communication between networked devices, allowing for reliable data transfer between hosts. In the **TCP/IP architecture**, every data request and transfer is tied to the IP address of the source and destination, meaning that the network is focused on the location of content (i.e., where it is hosted), rather than the content itself [8].

TCP/IP was originally designed for direct host-to-host communication, making it effective for earlier internet use cases like file sharing and email. However, as the internet evolved, the majority of traffic shifted toward **content distribution**, especially with the rise of video streaming platforms like **YouTube** and **Netflix**. This shift has revealed several limitations in the TCP/IP model, particularly related to **latency**, **bandwidth consumption**, and **scalability** [9], [10].

2.1.2 Video Streaming in TCP/IP Architectures

In TCP/IP-based networks, video streaming works by transmitting data between a video server (or CDN) and the client device (such as a computer, smartphone, or smart TV). The process involves several key components that ensure that video content is delivered efficiently and adapts to varying network conditions [9], [10].

At its core, TCP/IP uses two essential layers: the **Internet Protocol (IP)** for routing packets and the **Transmission Control Protocol (TCP)** for ensuring reliable delivery. Additionally, for video streaming, application-layer protocols such

as **HTTP** (HyperText Transfer Protocol) and **RTSP** (Real-Time Streaming Protocol) play critical roles [11]. Figure 2.1 showcases the journey which a video goes on when it has to be streamed from a site to a viewer.

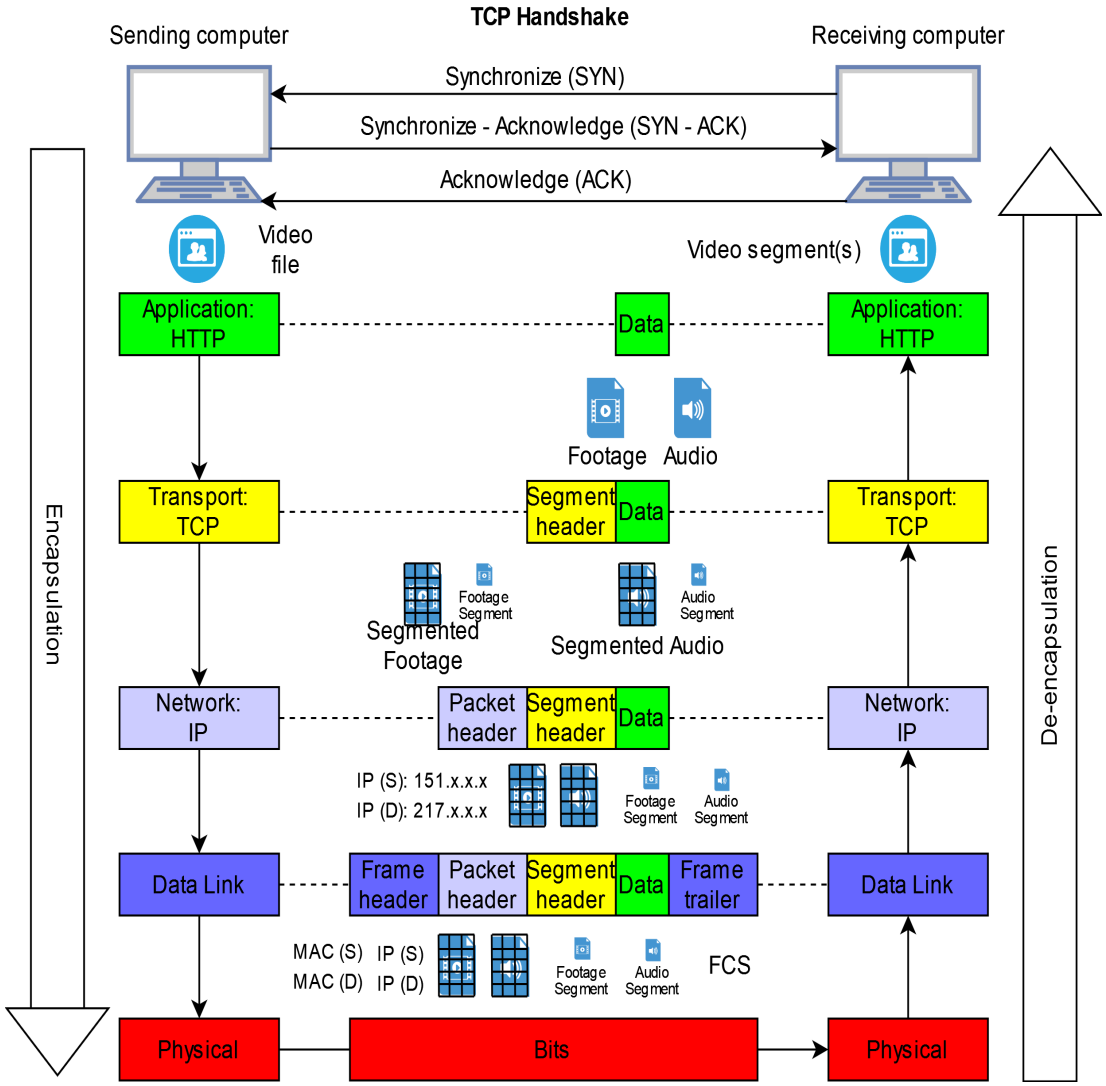

Figure 2.1: Diagram showing the sequence of events for a video file to go from a streaming site to a viewer (with chosen protocols to match real-world applications).

Application Layer: The video server, typically using HTTP, hosts the segmented video file, divided into separate audio and video streams. The client initiates streaming by sending an HTTP request to the server, which responds by streaming these segments back [9], [10], [12].

Transport Layer: TCP splits each video and audio segment into smaller packets, adding segment headers and ensuring reliable delivery through a three-way handshake (SYN, SYN-ACK, ACK). It attaches sequence numbers and checksums to handle errors, retransmitting lost packets as needed [8], [11], [13].

Network Layer: The IP protocol assigns source and destination IP addresses to each packet, guiding them through the network. Packets may take multiple paths to the client, enabling efficient routing [8], [14].

Data Link Layer: Packets are wrapped within frames, which include MAC addresses for both source and destination, as well as error-checking codes, like the Frame Check Sequence (FCS), to detect transmission errors [15]–[17].

Physical Layer: Frames are converted to binary bits, which are transmitted over physical media, such as Ethernet or Wi-Fi, facilitating the actual data transfer [9], [10].

De-encapsulation at the Client:

When the client receives the video data, each segment undergoes a de-encapsulation process at each layer. Starting at the Physical Layer, the raw bits are converted into frames by the Data Link Layer, which then checks for errors using the Frame Header and Trailer (containing the source and destination MAC addresses and error-checking codes) [14]. These headers and trailers are removed, and the data moves to the Network Layer, where the packet headers with IP addresses are stripped away [8]. At the Transport Layer, the segment headers, which include sequence numbers and error-checking information, are removed, ensuring that segments are correctly ordered and complete [8].

Finally, the data reaches the Application Layer, where it is reassembled into audio and video segments based on the manifest file. This reassembly allows the media player to begin buffering and rendering content in the correct sequence, enabling a smooth, continuous playback experience. The process repeats for each segment received, allowing for real-time adjustments in cases of adaptive streaming, where quality may change based on network conditions [18].

2.1.3 Packet Transmission in TCP/IP

In the TCP/IP architecture, video data is divided into smaller units called **packets** to facilitate reliable transmission. Each packet carries a portion of the video or audio data, with an IP header containing the source and destination IP addresses, and a sequence number allowing reassembly in the correct order at the destination [8]. This process, known as packetization, ensures that even high-bandwidth video files can be transmitted efficiently across networks with varying capacities [9], [10].

TCP, responsible for transport-layer functions, guarantees reliable delivery of packets using mechanisms such as error checking, retransmissions of lost packets, and flow control to regulate data transmission rates [8], [13]. These mechanisms are particularly critical in video streaming, where even a single lost or corrupted packet can disrupt playback quality [8], [13]. TCP's error-checking functions use checksums to detect errors, while retransmissions compensate for lost packets, ensuring complete and accurate data delivery [8], [11].

However, TCP's focus on reliability can introduce latency, which poses challenges in real-time video streaming. Each packet must be acknowledged by the receiver, and any lost packets require retransmission, which may delay playback or cause buffering. In scenarios with high packet loss or network congestion, TCP's retransmission and flow control mechanisms may slow down data transfer, impacting the Quality of Experience (QoE) for users [14].

In practice, streaming platforms implement adaptive bitrate streaming to manage the effects of TCP's packet transmission characteristics. By adjusting the video quality based on available bandwidth and packet delivery rates, adaptive streaming

mitigates TCP's latency and buffering issues. [18] For example, if the network is congested, the client automatically switches to a lower-quality stream that requires fewer packets, reducing the likelihood of interruptions. This dynamic adaptation is essential for smooth streaming experiences, as it allows the video player to balance playback quality and stability in response to changing network conditions [18].

The packet transmission process continues iteratively, with packets reaching the client and undergoing reassembly. When all necessary packets arrive, the data is recompiled into video and audio segments, which are then processed by the video player to create a seamless viewing experience. This iterative cycle of packetization, error-checking, and adaptive quality adjustments underscores the complexities of delivering high-quality video over TCP/IP networks.

2.1.4 Role of HTTP in Video Streaming

Most video streaming platforms like **YouTube**, **Netflix**, and **Hulu** use **HTTP-based adaptive streaming protocols**, such as **HLS (HTTP Live Streaming)** and **MPEG-DASH**, to deliver video content over TCP/IP. In these protocols, video files are stored on servers and delivered to clients in small segments via the widely-used HTTP protocol. This design leverages existing web infrastructure, including Content Delivery Networks (CDNs), to distribute content globally with scalability and efficiency [9], [10].

The typical HTTP-based adaptive streaming process operates as follows:

1. The user selects a video, prompting the client to send an HTTP request to the server.
2. The server responds by delivering the video in chunks, each typically lasting a few seconds.
3. Playback begins after downloading and buffering the initial segments, while subsequent chunks download in the background.
4. As playback continues, the client periodically requests additional segments from the server to maintain the buffer.

This **pull-based model** allows clients to retrieve content as needed based on playback progress. However, under network congestion, this approach may lead to buffering issues if the download rate falls behind the playback rate [12].

2.1.5 Adaptive Streaming: HTTP Adaptive Streaming (HAS)

To mitigate the limitations of basic HTTP streaming, **HTTP Adaptive Streaming (HAS)** was developed. HAS dynamically adjusts video quality based on the client's available bandwidth and network conditions, enhancing the viewing experience during fluctuations in performance [18].

In HAS, video content is segmented into small chunks, each encoded at multiple quality levels (bitrates). The process begins with the client fetching a manifest file (e.g., **M3U8** for HLS or **MPD** for MPEG-DASH), which lists available video

qualities for each segment. The client uses this information to dynamically select the optimal bitrate for playback. If network bandwidth decreases, the client seamlessly switches to a lower bitrate to prevent buffering.

The HAS workflow can be summarized in three stages:

- **Initial request**: The client requests the manifest file and the first video chunk, which contains metadata about available resolutions and bitrates.
- **Buffering**: The client buffers a few seconds of video before playback begins and downloads additional chunks in real-time.
- **Adaptive quality selection**: The client monitors network speed and buffer status to select the most suitable bitrate for subsequent chunks, switching quality levels as necessary to maintain smooth playback.

This adaptive mechanism minimizes buffering and enhances the **Quality of Experience (QoE)** for users by dynamically responding to changes in network conditions. Metrics such as buffer length, video resolution, and start delay significantly influence QoE, and protocols like **MPEG-DASH** and **HLS** have emerged as industry standards for ensuring reliable streaming performance [9], [10], [18].

2.1.6 Role of UDP and QUIC in Streaming

While HTTP-based streaming protocols primarily operate over **TCP**, some streaming platforms, including **YouTube**, have explored the use of **UDP**-based protocols such as **QUIC**. QUIC, developed by Google, offers reduced connection establishment time and improved congestion control, making it particularly advantageous for mobile and high-latency networks [19]. YouTube, as a part of Google's ecosystem, has implemented QUIC to improve video streaming performance, leveraging its reduced latency and enhanced adaptability to varying network conditions [20].

However, for the majority of video-on-demand and adaptive streaming scenarios, **TCP remains dominant** due to its reliability and compatibility with existing HTTP/CDN infrastructure [20].

2.1.7 Video Buffering and Latency

Buffering is a crucial technique in video streaming that mitigates the impact of network latency and packet loss. When a video is requested, the client initially buffers a portion of the video data before playback begins. This pre-buffering creates a "cushion" that helps maintain smooth playback even if brief network disruptions occur [15]–[17].

However, if the buffer is depleted due to slow data transfer rates, users experience playback interruptions, such as pauses or stuttering. High network latency or packet loss can lead to extended buffering times, directly impacting the **Quality of Experience (QoE)**. Metrics such as buffering frequency and duration are critical indicators of QoE and play a significant role in user satisfaction and platform retention [8], [14].

2.1.8 Challenges with TCP/IP in Video Streaming

TCP/IP's host-centric design presents several limitations for large-scale video streaming applications:

Centralized Content Delivery Video content is typically hosted on centralized servers. During peak times or popular events, these servers face heavy demand as multiple users request the same content simultaneously. This centralized approach can lead to bottlenecks, even when supported by **Content Delivery Networks (CDNs)**. For example, live-streamed sports events or viral videos may strain server resources, potentially resulting in buffering or degraded quality for users [15]–[17].

Redundant Data Transfers In TCP/IP-based streaming, each user independently retrieves the same video content, leading to significant redundancy in data transmission. For popular videos, this results in identical data being sent multiple times across the network, consuming excessive bandwidth and increasing network congestion [14].

Scalability Issues As the demand for video streaming grows, TCP/IP networks face increased scalability challenges. Each additional user adds to the load on network resources, often resulting in congestion, longer video load times, and reduced quality. This limitation stems from TCP/IP's end-to-end communication model, which does not inherently optimize for the repetitive nature of content requests seen in video streaming [14], [17], [21].

Latency and Bandwidth Bottlenecks TCP/IP's reliance on centralized servers can result in high **latency** and suboptimal **bandwidth usage**, especially when users are located far from the hosting server. Users in remote geographic locations may experience delays in data delivery, leading to longer buffering times and a degraded Quality of Experience (QoE) [14]. CDNs attempt to alleviate these issues by distributing content to geographically closer servers, but they do not completely eliminate the inherent limitations of the TCP/IP model.

2.1.9 Content Delivery Networks (CDNs)

Content Delivery Networks (CDNs) are widely used to improve video streaming by addressing TCP/IP's latency and scalability limitations. CDNs distribute content across geographically dispersed **edge servers**, minimizing the distance data travels from server to user and reducing network congestion. However, because CDNs still operate over TCP/IP, they cannot fully eliminate issues like redundant data transfers and centralized data control[14]–[17].

In the traditional TCP/IP model, all content requests route traffic to a central server, creating bottlenecks during peak demand. By contrast, CDNs replicate popular content on edge servers closer to end-users, which reduces latency and lowers server load by directing user requests to nearby cache locations.

Figure 2.2 illustrates the difference between non-CDN and CDN-based data distribution. In a non-CDN setup, traffic is directed to a single origin server, increasing load and latency. A CDN network, however, uses distributed edge servers to cache content locally, reducing load on the origin server and improving delivery speed and reliability.

Although CDNs enhance scalability, they introduce complexities in **cost**, **maintenance**, and **optimization**. Maintaining multiple replicas across locations requires substantial resources, and CDNs still rely on a **pull-based model**, fetching data only upon request. This model can be inefficient when the same content is frequently requested from multiple locations [15]–[18].

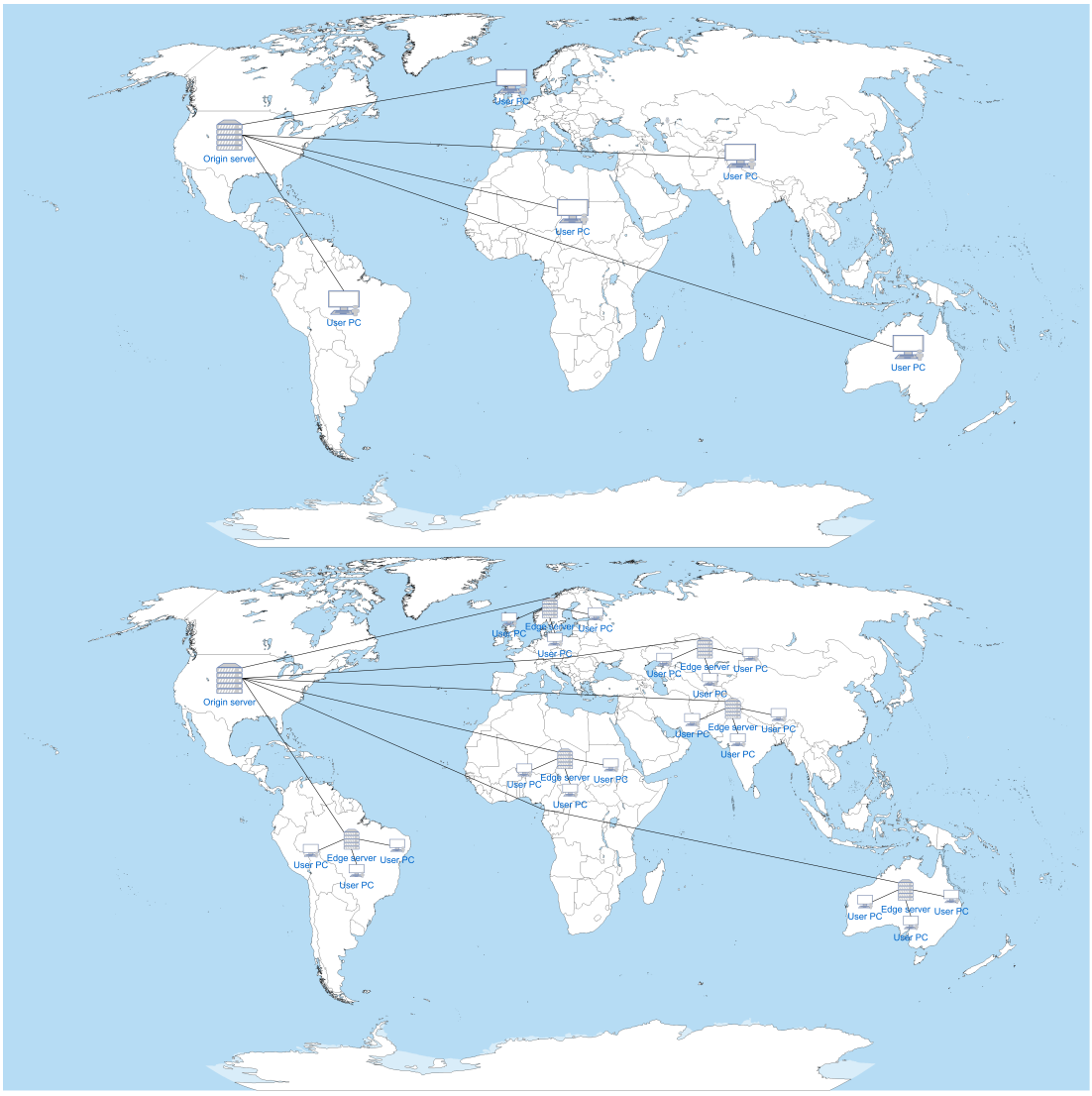

Figure 2.2: Comparison between a non-CDN and CDN-based network. In a non-CDN model, traffic is directed to the origin server, creating bottlenecks. A CDN structure, with geographically distributed edge servers, reduces latency and load on the origin server by providing users with cached content from nearby locations.

2.2 Named Data Networking (NDN)

2.2.1 Introduction to NDN

Named Data Networking (NDN) represents a paradigm shift from the conventional *host-centric* model of the current Internet, as exemplified by the Transmission Control Protocol/Internet Protocol (TCP/IP) architecture, toward a *content-centric* or *data-centric* model [2], [22], [23]. In traditional IP-based networks, communication revolves around the concept of endpoints identified by IP addresses. Users must know the location of the content (i.e., which server hosts it) before retrieving it, inherently coupling data availability and efficiency to the physical distribution of servers and infrastructure [2], [8].

NDN, in contrast, enables users to request content by *name*, abstracting away the physical location of data and focusing on what data is needed rather than where it resides [2], [22], [24]. By decoupling content from specific hosts, NDN seeks to address fundamental limitations of the current Internet architecture, particularly in scenarios where multiple users request the same content simultaneously [23]. Under traditional paradigms, such as when a widely viewed video on YouTube experiences a surge in popularity, each user's request often traverses a potentially congested path back to a central server or CDN node. This process increases latency, consumes bandwidth, and places a significant load on origin servers [14].

Through its *content-centric* design, NDN aims to improve network efficiency, scalability, and security . Because data in NDN is addressed by name rather than by host location, the network can retrieve it from intermediate caches rather than always relying on the original server [23]. This capability becomes increasingly important as user demands shift toward content-heavy services like high-definition video streaming and virtual reality, where efficient, low-latency data retrieval is critical [21], [25].

Moreover, NDN integrates *in-network caching* directly into its architecture, allowing content to be temporarily stored at multiple points along delivery paths. When new requests for the same content are issued, these can be served from a nearby cache instead of a distant origin, drastically reducing both latency and bandwidth usage [5], [22], [25]. By leveraging these caches, NDN naturally supports multicast-like behavior for popular content, reducing redundant transmissions and improving the overall resource utilization of the network [4], [25].

Beyond performance benefits, NDN's design also offers inherent advantages in terms of robustness and security. Its data-centric approach allows for content authentication at the data object level, rather than depending solely on securing point-to-point communication channels. This opens up possibilities for more flexible security and trust models [24].

2.2.2 Differences Between NDN and TCP/IP Architectures

The most fundamental distinction between Named Data Networking (NDN) and the traditional TCP/IP architecture lies in the core principle by which data is identified, requested, and delivered. In TCP/IP, communication is inherently *host-centric*,

meaning that data retrieval is tied to the location of a specific host identified by an IP address [8]. Clients must explicitly establish a connection to a remote server or CDN node, and the network's primary task is to route packets from the sender's IP address to the receiver's IP address. This approach, while effective during the early stages of the Internet's evolution, becomes increasingly inefficient in scenarios where popular content is requested by large, geographically dispersed audiences [21].

In contrast, NDN employs a *content-centric* approach that dissociates data from its physical storage location. Instead of specifying where data resides, requests in NDN identify *what* data they need by using *content names*—hierarchical, often human-readable labels that uniquely reference a piece of content (e.g., `/videos/travel/italy.mp4`). When a user issues an *Interest* packet containing such a name, NDN routers forward the request toward any node that can provide the data. If an intermediate node has cached the requested content, it can return a *Data* packet immediately, bypassing the need to reach an origin server [2], [22].

This shift in paradigm confers several advantages:

- **Location Independence:** Users no longer require knowledge of a server's address. The network naturally selects appropriate paths to sources or caches holding the requested content, reducing latency and simplifying the user experience.
- **Adaptive Routing and Caching:** Unlike TCP/IP, which relies on relatively static routing entries and pre-defined infrastructure like CDNs, NDN's name-based routing and in-network caching enable the network to dynamically adapt to changing demand patterns and network conditions [4], [21].
- **Enhanced Efficiency:** Content that is frequently requested can be cached closer to end-users, dramatically cutting down on redundant transmissions, server load, and congestion, resulting in better Quality of Experience (QoE) [21].
- **Simplified Multicast:** In TCP/IP, delivering identical content to multiple users typically requires separate unicast streams or complex multicast protocols. NDN's architecture inherently supports multicast-like behavior since the same cached data can satisfy multiple Interests, reducing complexity and overhead [21].

By focusing on *what* is being requested rather than *where* it resides, NDN eliminates the need for clients to engage in endpoint-based negotiation and overhead. In doing so, it paves the way for more flexible, resource-efficient, and user-oriented content distribution models.

2.2.3 In-Network Caching

A defining characteristic of Named Data Networking (NDN) is its integral support for *in-network caching*, wherein intermediate routers temporarily store copies of recently retrieved data [3]–[5]. This design stands in contrast to the traditional IP-based framework, where content retrieval generally depends on repeatedly fetching

data from distant origin servers or specialized Content Delivery Network (CDN) nodes [2], [22]. By allowing data to be cached at multiple points in the network, NDN enhances content availability, reduces latency, and alleviates bandwidth pressure on origin servers [4], [21], [22].

In a TCP/IP environment, a user's request for a popular video or dataset must typically travel upstream to a designated server or CDN edge node, consuming valuable time and resources. Under high-demand scenarios—such as viral videos, popular news clips, or frequently accessed software updates, this host-centric retrieval model can lead to congestion, increased latency, and excessive redundancy in data delivery [21]. Even when CDNs are employed, their caching points are relatively static and require explicit prepositioning of content.

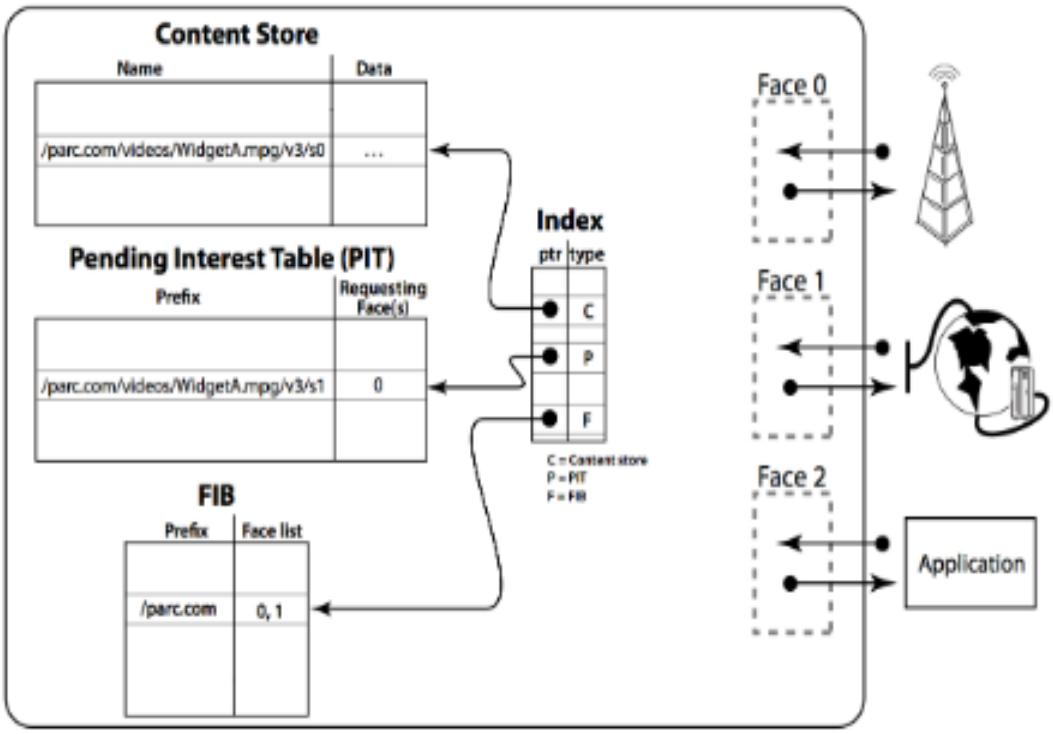

Figure 2.3: Forwarding process at an NDN node, detailing how Interest packets are routed and satisfied via the Content Store (CS), Pending Interest Table (PIT), and Forwarding Information Base (FIB). Cached Data packets reduce latency and bandwidth usage [23].

NDN's caching model, by contrast, is both dynamic and pervasive. Whenever a router forwards a *Data* packet in response to an *Interest*, it may store a copy of that data in its *Content Store (CS)*. Subsequent requests for the same content can then be satisfied locally from the router's CS, without the need to query distant sources [5], [25]. This reduces the load on the network's core and lowers retrieval times for end-users [21]. Numerous empirical and simulation-based studies have demonstrated that in-network caching can significantly reduce bandwidth consumption and improve scalability, particularly for content distribution applications [3]–[5], [25].

Furthermore, in-network caching facilitates a form of implicit multicast. Instead of serving each user's Interest independently from a centralized repository, the same cached copy can fulfill multiple downstream requests. This approach naturally aligns with real-world usage patterns, where popular content tends to be requested by many users over short time frames [21]. As a result, NDN's in-network caching not only enhances performance for individual consumers but also contributes to global network efficiency and fairness.

2.2.4 Data Naming and Content Routing

At the core of Named Data Networking's paradigm shift lies the concept of *data naming*—the ability to identify content by a human-readable, hierarchical name rather than by the location of a host. In contrast to TCP/IP networks, where data retrieval is predicated on knowing an endpoint's IP address, NDN's naming scheme enables users to express their interests in the data directly, such as `/videos/travel/italy.mp4`, regardless of where that data might reside [2], [22].

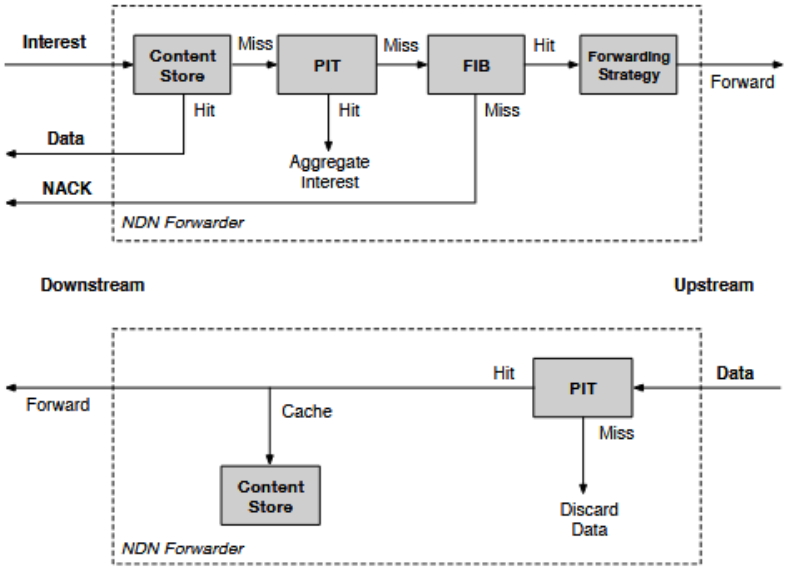

Figure 2.4: Node-level architecture in NDN, showing how the Content Store (CS), Pending Interest Table (PIT), and Forwarding Information Base (FIB) collaborate to efficiently handle Interest and Data packets [23].

This naming approach fosters a more flexible and semantically rich retrieval process. Each name can be composed of multiple components, often delineated by slashes, allowing the network to treat each component as a meaningful element of a hierarchical namespace. Such hierarchical names not only facilitate efficient lookups and aggregations but also permit fine-grained control and caching of data at various levels of the naming hierarchy [21], [26].

In NDN, data retrieval is orchestrated through the exchange of two types of packets: *Interest* and *Data*. When a user's application seeks a specific piece of content, it sends out an Interest packet containing the content's name. Routers along the path use their *Forwarding Information Base (FIB)*—populated by name-based routing protocols—to forward the Interest toward potential sources of the

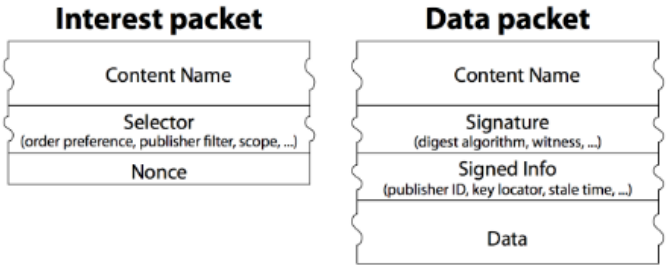

Figure 2.5: Structure of Interest and Data packets in NDN. The Interest packet contains fields such as *Content Name* and *Nonce*, while the Data packet includes *Signature* and *Signed Info*, ensuring secure and precise content delivery [23].

data. Importantly, no explicit connection or session establishment to a particular host is required. Instead, the network is free to find any suitable source, whether that be the original producer of the content or an intermediate cache [24], [27].

Upon discovering the requested data, the responding node sends back a *Data* packet along the reverse path of the Interest. This Data packet includes the requested content and carries the same hierarchical name, enabling routers to satisfy the Interest without ambiguity. Along its return path, the Data packet may be cached at routers for potential future Interests, thus exploiting NDN's in-network caching capabilities [21]. The result is a self-regulating and loop-free retrieval process that can gracefully adapt to changes in network topology, traffic loads, and content popularity.

The *Pending Interest Table (PIT)* within each NDN router tracks Interests that have been forwarded upstream but not yet satisfied. When a matching Data packet arrives, the router consults its PIT to identify which downstream interfaces requested the content and forwards the Data accordingly, removing the corresponding PIT entries. This mechanism enables Interest aggregation—if multiple users request the same piece of content around the same time, only one Interest needs to travel upstream, reducing redundant transmissions and network congestion [22], [27].

In essence, data naming and content routing in NDN decouple data retrieval from the underlying network topology and host identifiers. By shifting focus from *where* data is located to *what* data is desired, NDN achieves a more flexible, scalable, and user-centric model of content dissemination.

2.2.5 Security Through Data-Centric Design

Unlike traditional IP-based networks—where security measures often focus on protecting communication channels between specific endpoints—Named Data Networking (NDN) employs a *data-centric* security model. In the conventional TCP/IP paradigm, protocols such as TLS/SSL aim to establish secure sessions, ensuring that data in transit cannot be easily intercepted or tampered with [28]. However, once data leaves the secured connection, its integrity and authenticity rely heavily on the trustworthiness of intermediate systems, and any caching or content redistribution mechanisms must be carefully managed to prevent content spoofing or corruption.

NDN, on the other hand, integrates security directly into the content itself. Every piece of data is cryptographically signed by its producer, providing a signature that can be verified by any recipient, regardless of where or how the data is obtained [2], [24], [26]. This data-centric approach means that trust and authenticity are inherently bound to the data objects, rather than the channels over which they travel. Even if data is cached at intermediate routers, end-users can verify signatures to confirm the provenance and integrity of the received information. As a result, content can safely be retrieved from potentially untrusted caches, allowing NDN to realize the full benefits of widespread in-network storage without sacrificing security.

This shift in perspective simplifies many aspects of secure content distribution. For example, consider a popular video segment cached across multiple nodes: in NDN, any consumer requesting that segment can validate its authenticity by checking the attached signature. There is no need to re-establish secure channels, renegotiate keys, or trust that the caching entities are honest. Instead, verifiable signatures provide intrinsic protection against malicious alterations or spoofed content [24], [26].

Furthermore, NDN's name-based architecture supports flexible trust models and hierarchical trust relationships. Public keys can be named and retrieved just like other data objects, and trust schemas can be defined to specify which keys are authorized to sign which types of content. This enables more fine-grained and context-dependent trust decisions, as well as simplified key distribution and revocation mechanisms [29], [30].

Although this approach addresses many security concerns at the data object level, it does not eliminate all possible attacks. For instance, NDN remains susceptible to Interest flooding attacks, where adversaries send large volumes of Interests for non-existent or rarely requested data, potentially overwhelming router resources [24], [26]. Ongoing research focuses on developing robust countermeasures, congestion control strategies, and more sophisticated trust management frameworks to further enhance the security and resilience of NDN-based networks.

2.2.6 Cache Placement and Replacement Policies

While the principle of in-network caching is central to Named Data Networking (NDN), its effectiveness depends heavily on the policies governing which content is cached, where it is placed, and how long it remains stored. The goal is to maximize cache utility—ensuring that frequently requested data is readily available—while minimizing redundancy and unnecessary resource consumption [4], [21]. Two key dimensions define NDN caching strategies: *cache placement* (when and where to cache data along the delivery path) and *cache replacement* (which items to evict when cache capacity is limited).

Cache Placement Policies By default, NDN follows a *Leave Copy Everywhere (LCE)* policy, in which each router that forwards a *Data* packet caches a copy of it. While LCE ensures that content is quickly disseminated throughout the network, it can lead to significant redundancy, wasting cache space and bandwidth [21]. To address this inefficiency, several alternative placement strategies have been proposed:

- **Leave Copy Down (LCD):** Caches content only at the first router encountered downstream of a hit, gradually moving the content closer to consumers over time. This reduces upstream redundancy while still bringing data closer to areas of high demand [4].
- **Move Copy Down (MCD):** Similar to LCD, but instead of copying the data, the cache is effectively "moved" downstream on a hit. This approach aims to maintain a single copy closer to the consumer, further reducing duplication [3].
- **Probabilistic Caching (ProbCache):** Assigns a probability to caching decisions, thereby avoiding deterministic placement at every hop. For instance, "Copy with Probability (p)" strategies cache content with a certain probability at each router, balancing redundancy and accessibility. More sophisticated probabilistic schemes (e.g., ProbCache) consider factors like the router's distance to the consumer or the expected number of downstream copies to optimize caching efficiency [3].

These placement strategies aim to reduce redundant caching, concentrate content where it is most needed, and leverage the inherent adaptability of NDN's architecture.

Cache Replacement Policies Even with intelligent placement strategies, caches have finite capacity. As they become full, decisions must be made about which items to retain. Classical replacement policies from traditional caching literature are also applicable to NDN, though their performance may differ given the unique dynamics of named-data retrieval [5], [25], [31]:

- **Least Recently Used (LRU):** Evicts the oldest accessed item first, assuming that recently requested items are more likely to be popular in the near future [31].
- **Least Frequently Used (LFU):** Tracks the frequency of accesses and discards the least often requested items, favoring persistently popular content [31].
- **First-In-First-Out (FIFO):** Removes items in the order they were stored, independent of access patterns, thus providing a simple but often suboptimal solution [31].
- **Random Replacement:** Evicts a randomly selected item, occasionally useful as a baseline or in scenarios where system overhead must be minimized [31].

Researchers have explored more advanced replacement policies that integrate popularity predictions, cost-aware decisions, or latency constraints, aiming to align cache management more closely with network-wide objectives [3].

Balancing Efficiency and Overhead Selecting optimal placement and replacement policies involves trade-offs. More complex strategies may offer better performance but can introduce additional overhead in terms of state tracking, computation, or memory usage. Consequently, network operators and system designers must weigh the benefits of improved cache hit ratios and reduced latency against the complexity and scalability challenges that more sophisticated policies may impose [31].

2.2.7 Adaptive Forwarding Strategies

While the fundamental mechanisms of naming, caching, and data retrieval form the core of Named Data Networking (NDN), the architecture also accommodates dynamic and *adaptive forwarding strategies* to optimize data delivery under varying network conditions. Traditional IP-based routing typically relies on predetermined best-path forwarding, established through static routing protocols or limited dynamic adjustments. In contrast, NDN forwarding strategies are designed to monitor and respond to factors such as link quality, congestion, and content availability in near-real time, thereby improving the scalability, reliability, and efficiency of data dissemination [5], [24], [32].

Motivation for Adaptive Forwarding NDN's reliance on *Interest/Data* exchanges, combined with in-network caching, creates opportunities for more nuanced forwarding decisions than simply sending packets along a single best path. For example, if multiple upstream nodes hold cached copies of requested content, an NDN router can probe different paths and select the one that offers the shortest round-trip time, least congestion, or highest cache hit probability [3], [5]. This adaptiveness ensures that NDN can exploit the network's inherent redundancy and flexibility, leading to more efficient load distribution and faster recovery from link failures.

Forwarding Architecture and Data Structures To implement adaptive forwarding, NDN nodes maintain key data structures: the *Forwarding Information Base (FIB)*, the *Pending Interest Table (PIT)*, and the *Content Store (CS)*. While the PIT and CS handle Interest/Data matching and caching, respectively, the FIB provides multiple potential next-hops for given name prefixes. The forwarding strategy logic within each node leverages the FIB's multiple entries to attempt alternate paths if the first choice underperforms. This logic may include metrics such as Smoothed Round-Trip Time (SRTT), packet loss rates, or observed content hit ratios [27], [32], [33].

Examples of Adaptive Strategies A range of forwarding strategies have been proposed and evaluated in the literature:

- **Best Route Strategy:** A baseline approach that forwards Interests along a single best-known path. Although simple, this strategy lacks adaptability in dynamic scenarios [27].

- **Multicast or Parallel Forwarding Strategies:** Forwarding an Interest simultaneously along multiple paths, then choosing the path that returns data first. Subsequent Interests may then favor that path, improving responsiveness [5].
- **Measurement-Based Adaptation:** Strategies that maintain statistical measures (e.g., SRTT, interface utilization, cache hit probability) to rank upstream interfaces and forward Interests preferentially through the best-performing paths, continuously adjusting these choices as network conditions evolve [3], [5].
- **Machine Learning and Heuristic-Based Approaches:** Recent work explores integrating learning algorithms to predict which paths will likely yield the best performance, using historical data, traffic patterns, and content popularity [34], [35].

Challenges and Trade-Offs Adaptive forwarding is not without challenges. Overly aggressive probing of multiple paths can introduce unnecessary overhead and complexity. Striking a balance between exploration (trying new paths) and exploitation (using the best-known path) is essential to avoid oscillations and instabilities in routing decisions. Additionally, while adaptive forwarding can improve QoE and resource utilization, it may also increase router state complexity and the need for more sophisticated forwarding engines [21].

2.2.8 Video Streaming in NDN

One of the most prominent application domains for Named Data Networking (NDN) is video streaming—an area of immense growth and critical importance in today's Internet. As video content increasingly dominates network traffic, traditional TCP/IP-based delivery models often struggle to provide consistently low latency, high bandwidth efficiency, and robust Quality of Experience (QoE). NDN's *content-centric* architecture, *in-network caching*, and *adaptive forwarding strategies* collectively offer new opportunities to meet these challenges [21].

Leveraging In-Network Caching for Video Segments In typical video-on-demand and live streaming scenarios, popular segments of media content are repeatedly requested by geographically dispersed users within short time intervals. NDN naturally addresses this pattern by caching data packets closer to the user, effectively reducing round-trip times and smoothing out traffic spikes. Instead of fetching every requested segment from a distant origin server or CDN node, routers along the delivery path can satisfy subsequent Interests from local caches, decreasing bandwidth consumption and latency [4], [5], [25].

Adaptive Bitrate and Segment-Based Delivery Modern video streaming protocols, such as Dynamic Adaptive Streaming over HTTP (DASH) and HTTP Live Streaming (HLS), rely on segment-based delivery and adaptive bitrate selection to match video quality to network conditions [18]. NDN's named data model aligns well with these segment-based architectures by enabling efficient caching and

retrieval of individual segments [18], [21]. Each video segment can be named hierarchically, for example:
`/videos/tutorial/session1/segment1/bitrate2Mbps`, enabling fine-grained caching and retrieval. NDN's adaptive forwarding and in-network caching can complement segment-based strategies by ensuring that high-demand segments at popular bitrates are readily available, thereby improving QoE during transient congestion or link failures [5], [18].

Popularity-Aware Prefetching and QoE Enhancement Since NDN inherently supports retrieving data from multiple sources (original producers or caches), video streaming applications can leverage user consumption patterns to predict which segments are likely to be requested next. Popularity-based prefetching—where frequently accessed segments are proactively cached or retained at strategic locations—helps mitigate startup delays and buffering events. By integrating user behavior metrics, such as segment skip patterns or re-watch frequencies, NDN-based video delivery can further reduce unnecessary data transfers and optimize caching strategies, ensuring that the most in-demand portions of content are immediately accessible [6], [7].

2.3 Prefetching Methods

Prefetching is a proactive technique used to predict and fetch content before a user explicitly requests it. This approach enhances **Quality of Experience (QoE)** by minimizing perceived latency and buffering while improving network efficiency [36]. Prefetching methods can be broadly categorized into three main types: rule-based, popularity-based, and history-based approaches.

2.3.1 Rule-Based Prefetching

Rule-based prefetching operates using predefined heuristics to anticipate user requests. A common strategy involves fetching the next sequential video segment immediately after the current one is delivered. While simple and efficient in predictable scenarios, these methods lack adaptability to dynamic network conditions, leading to potential bandwidth inefficiencies in variable environments [36].

2.3.2 Popularity-Based Prefetching

Popularity-based approaches leverage statistical analysis of content popularity metrics such as view counts, click-through rates, and watch times to prioritize fetching highly demanded content. This method is particularly effective for popular videos, live streams, or trending topics, where user behavior is relatively predictable. Prefetching based on popularity metrics can substantially reduce latency for widely accessed content [36].

2.3.3 History-Based Prefetching

History-based prefetching utilizes patterns of user behavior over time to predict future requests. Techniques in this category include the use of dependency graphs,

Markov models, and data mining algorithms. For instance, Markov-based models analyze sequences of previously accessed content to estimate the likelihood of future requests. Similarly, data mining approaches use clustering and association rule mining to identify frequently co-accessed segments, optimizing prefetching decisions [36].

2.3.4 Challenges and Trade-Offs

Although prefetching techniques significantly improve QoE, they also introduce challenges, such as:

- **Overfetching:** Excessive prefetching can lead to bandwidth waste and storage inefficiencies [36].
- **Adaptability:** Predictive models must adapt to rapidly changing user behavior and network conditions [36].
- **Privacy Concerns:** Advanced predictive methods often require extensive data collection, raising potential privacy issues [36].

2.3.5 Prefetching in NDN

NDN provides a content-centric approach to prefetching by decoupling content from its storage location. In NDN, prefetching can leverage in-network caching and content popularity to optimize data delivery. By predicting high-demand segments and proactively storing them closer to users, NDN-based prefetching offers significant advantages over traditional methods. These include reduced latency, improved scalability, and lower bandwidth consumption [36], [37].

Chapter 3

Concept Design

In designing a prefetching mechanism for video streaming in NDN, the primary objective is to enhance user experience by reducing latency and buffering while efficiently utilizing network resources. The focus of this project is on implementing a popularity-based prefetching strategy that identifies high-demand video segments and proactively caches them closer to end-users. By leveraging user engagement data extracted from video heatmaps, the proposed mechanism predicts and fetches content segments likely to be requested next, optimizing content delivery.

The prefetching strategy is designed to integrate seamlessly with NDN's existing content-centric architecture. It does so by leveraging the ability of routers to analyze content popularity and adapt their caching behavior dynamically. Instead of relying solely on reactive retrieval, where content is fetched only upon user request, the proposed mechanism proactively retrieves and stores popular video segments in anticipation of future requests. This design choice minimizes reliance on real-time data transmission, ensuring uninterrupted content retrieval even in scenarios involving network congestion or link failures.

The core idea centers around analyzing user behavior to determine which parts of a video are more likely to be accessed. Using engagement heatmaps, the system identifies popular segments based on metrics such as watch time or user skips. These segments are then prefetched and cached closer to the network edge, reducing the latency associated with fetching these segments from distant content producers. The mechanism is implemented without introducing additional complexity to NDN's forwarding strategies or user-side applications, ensuring compatibility with the broader ecosystem.

To evaluate the proposed mechanism, simulations are conducted using a range of scenarios that compare the performance of the popularity-based prefetching strategy against alternative caching and delivery approaches. These scenarios are designed to highlight the strengths and limitations of the method in various network conditions. The details of these scenarios are discussed in the implementation section under Simulation.

This project aims to provide a practical and scalable solution for video streaming, addressing key challenges in content delivery while maintaining simplicity and adaptability. By focusing on the integration of popularity metrics into caching decisions, the proposed mechanism represents a step forward in leveraging user behavior analytics for efficient and user-centric content delivery.

Chapter 4

Implementation

4.1 Simulation Overview

To evaluate the effectiveness of the proposed prefetching mechanism, four distinct scenarios were implemented and simulated within the NDN environment. These scenarios explore various configurations, ranging from a CDN-like setup to advanced prefetching mechanisms with caching strategies. The simulations were conducted using the `ndnSIM` tool, leveraging two distinct network topologies: one mimicking a traditional CDN configuration and another designed to represent a cloud-based producer topology.

Figures 4.1 and 4.2 illustrate the topologies used. Scenario 1 employs a topology optimized for a CDN, with producers located close to consumers to minimize latency. Scenarios 2-4 use an Abilene-inspired topology, with a single producer located farther away to emulate a cloud-based setup. These topologies were adapted from templates provided within the `ndnSIM` library and refined based on insights from prior research and related theses.

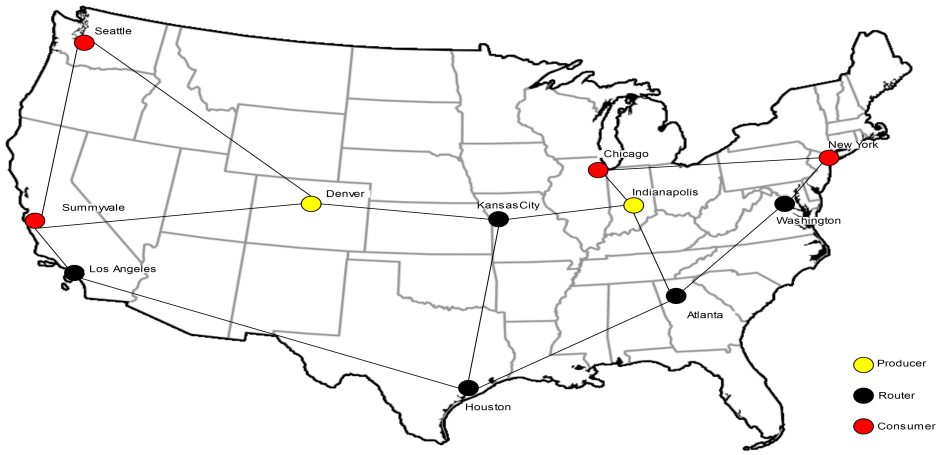

Figure 4.1: Topology used in Scenario 1, simulating a traditional CDN with producers positioned close to the consumers.

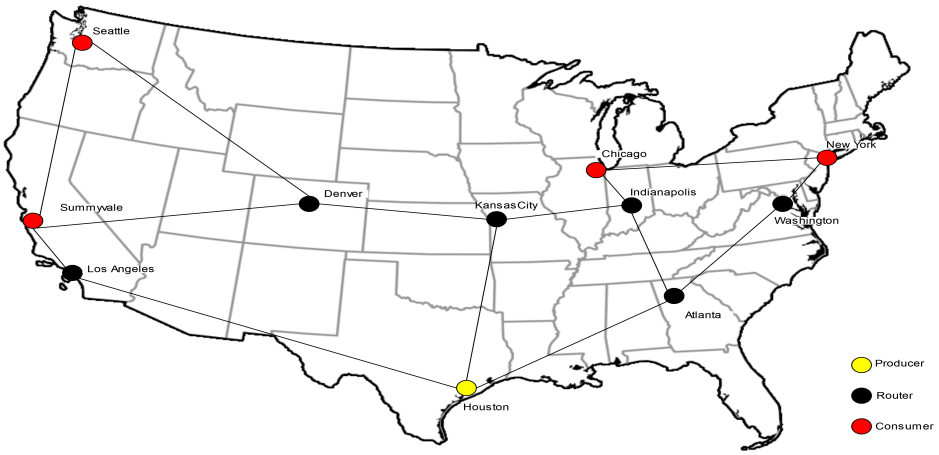

Figure 4.2: Topology used in Scenarios 2-4, simulating NDN-based content delivery with varied prefetching and caching strategies.

The scenarios were designed to evaluate the performance of the proposed mechanism under various configurations. These configurations include scenarios with no prefetching, basic caching mechanisms, prefetching without replacement policies, and prefetching integrated with caching strategies such as FIFO and LRU. To better reflect real-world usage patterns, consumer behaviors were also varied across simulations to include sequential viewing, skipping segments, random access, and popularity-based access patterns.

The results of these simulations were informed by data extracted from YouTube videos using a custom Python pipeline described in the following section. This data served as the foundation for generating realistic engagement patterns and segment popularity metrics, enabling a comprehensive evaluation of the proposed mechanism.

4.2 Data Extraction for Popularity-Based Prefetching

This section describes the implementation of the data extraction pipeline designed to support popularity-based prefetching in NDN. The pipeline consists of scripts that automate the retrieval, processing, and analysis of video engagement data, extracted from YouTube heatmaps, to identify high-engagement segments for effective prefetching.

4.2.1 Overview of the Pipeline

The pipeline is implemented as a single orchestrated script, `main.py`, which coordinates five distinct modules to achieve the following:

1. **Heatmap Extraction:** Downloads heatmap visualizations from YouTube videos in SVG format.
2. **SVG Modification and PNG Conversion:** Processes and converts SVG files into PNG images for pixel-based analysis.

3. **Engagement Data Extraction:** Analyzes the PNG images to compute viewer engagement metrics.
4. **Segment Popularity Calculation:** Computes the average engagement for predefined video segments and saves the results in JSON format.
5. **Engagement Visualization:** Generates a line plot of user engagement over time.

The pipeline enables structured and reproducible extraction of user engagement data, providing essential inputs for popularity-based prefetching algorithms.

4.2.2 Web Scraping and Heatmap Extraction

The first step in the pipeline involves extracting heatmaps from YouTube videos. This is achieved using the `extract_svgs_from_youtube()` function, which automates browser interactions via Selenium. The function opens the specified YouTube video, scrolls through the page to ensure all heatmaps are loaded, and injects JavaScript into the page to extract and download heatmaps as SVG files. By using the Firefox WebDriver, the function ensures seamless automation of web scraping tasks, while the injected JavaScript precisely identifies and serializes the SVG heatmap elements on the page. Finally, the SVG files are saved with appropriate formatting to facilitate subsequent processing.

```
def extract_svgs_from_youtube(video_url):
    """Extracts heatmaps by injecting JavaScript into the YouTube page using Selenium."""
    js_code = """
    (function() {
        const heatmaps = document.querySelectorAll('.ytp-heat-map-container svg');
        heatmaps.forEach((heatmap, index) => {
            heatmap.removeAttribute('width');
            heatmap.removeAttribute('height');
            if (!heatmap.hasAttribute('viewBox')) {
                const bbox = heatmap.getBBox();
                heatmap.setAttribute('viewBox', `0 0 ${bbox.width} ${bbox.height}`);
            }
            const svgData = new XMLSerializer().serializeToString(heatmap);
            const svgBlob = new Blob([svgData], { type: 'image/svg+xml;charset=utf-8' });
            const url = URL.createObjectURL(svgBlob);
            const downloadLink = document.createElement('a');
            downloadLink.href = url;
            downloadLink.download = `heatmap_${index + 1}.svg`;
            document.body.appendChild(downloadLink);
            downloadLink.click();
            document.body.removeChild(downloadLink);
            URL.revokeObjectURL(url);
        });
    })();
    """
```

```
    options = Options()
    profile = FirefoxProfile()
    profile.set_preference("browser.download.folderList", 2)
    profile.set_preference("browser.download.dir", SVG_SAVE_DIR)
    profile.set_preference("browser.helperApps.neverAsk.saveToDisk"
    , "image/svg+xml")
    options.profile = profile
    options.add_argument("--headless")
    driver = webdriver.Firefox(options=options)
    try:
        driver.get(video_url)
        time.sleep(5)
        driver.execute_script("window.scrollTo(0, document.body.
    scrollHeight);")
        time.sleep(3)
        driver.execute_script(js_code)
        time.sleep(5)
    finally:
        driver.quit()
```

Listing 4.1: Extracting SVG Heatmaps from YouTube

4.2.3 Modifying and Converting SVGs to PNGs

Once the SVG files are downloaded, they need to be modified and converted to PNG format for pixel-based analysis. The `modify_and_convert_svgs()` function handles these tasks. This function adjusts the color of SVG elements to grayscale to standardize the visual data and resizes the SVG dimensions to match the predefined video segment durations. It also utilizes `cairosvg` to convert the modified SVGs into high-resolution PNGs, ensuring compatibility with image analysis workflows.

```
def modify_and_convert_svgs(input_svg_folder, modified_svg_folder,
    output_png_folder):
    """Modify SVGs (color and size) and convert them to PNGs."""
    def change_svg_color_to_gray(svg_file):
        tree = ET.parse(svg_file)
        root = tree.getroot()
        for elem in root.iter():
            if 'fill' in elem.attrib:
                elem.set('fill', 'rgb(128,128,128)')
        return tree, root

    for i, (start, end) in enumerate(segments):
        segment_duration = end - start
        segment_width = calculate_segment_width(segment_duration,
    total_video_length)
        svg_file = os.path.join(input_svg_folder, f'heatmap_{i+1}.
    svg')
        png_file = os.path.join(output_png_folder, f'heatmap_{i+1}.
    png')
        tree, root = change_svg_color_to_gray(svg_file)
        if root.tag.split('}')[-1] == 'svg':
            root.set('width', str(segment_width))
            root.set('height', '100')
```

```
20              modified_svg_file = os.path.join(modified_svg_folder, f
    'modified_heatmap_{i+1}.svg')
21              tree.write(modified_svg_file)
22              cairosvg.svg2png(url=modified_svg_file, write_to=
    png_file, scale=4)
```

Listing 4.2: Modifying and Converting SVGs to PNGs

To ensure that each video segment's visual representation aligns proportionally with its allotted time in the overall video, the function `calculate_segment_width()` computes the width of each segment. This calculation ensures that each segment's representation accurately reflects its duration relative to the total video length.

The function uses the following formula:

$$\text{Segment Width} = \left(\frac{\text{Segment Duration (in seconds)}}{\text{Total Video Duration (in seconds)}} \right) \times \text{Base Width}$$

Why This Calculation Is Necessary

When processing heatmap SVG files, each segment of the video must be resized to visually represent its proportionate time duration. This resizing ensures that the engagement data extracted from the image corresponds accurately to the time intervals of the video. For example, a segment of shorter duration will take up proportionally less width compared to a longer segment.

This proportional allocation aligns the image data with the video timeline, ensuring accurate mapping of pixel-based engagement data to specific time intervals.

```
1 def calculate_segment_width(segment_duration, total_video_duration,
      base_width=1000):
2     total_seconds = total_video_duration.total_seconds()
3     segment_seconds = segment_duration.total_seconds()
4     return (segment_seconds / total_seconds) * base_width
```

Listing 4.3: Segment Width Calculation

4.2.4 Extracting Engagement Data from PNGs

The PNG files generated from the previous step are analyzed to extract engagement data. The `extract_engagement_data()` function maps pixel intensities in the PNG images to normalized engagement percentages and associates them with specific time points in the video. This function ensures accurate alignment between visual engagement data and video timestamps, outputting the results in a structured CSV format for further processing.

```
1 def extract_engagement_data(image_directory, csv_file_path):
2     """Extract pixel-based engagement data from PNG images."""
3     global_max_gray_value = 0
4     for image_path in image_directory:
```

```
        image = cv2.imread(image_path)
        gray_pixel_counts = count_gray_pixels(image)
        global_max_gray_value = max(global_max_gray_value, max(
    gray_pixel_counts))
    with open(csv_file_path, "w") as file:
        writer = csv.writer(file)
        writer.writerow(["Image", "Pixel", "Time (s)", "Gray Value"
    , "Normalized Engagement (%)"])
        for image_path in image_directory:
            image = cv2.imread(image_path)
            gray_pixel_counts = count_gray_pixels(image)
            normalized_gray_values = [(count /
    global_max_gray_value) * 100 for count in gray_pixel_counts]
            time_mappings = map_pixels_to_time(start_time, duration
    , width)
            for pixel_num, (time, gray, normalized) in enumerate(
    zip(time_mappings, gray_pixel_counts, normalized_gray_values)):
                writer.writerow([f"Image_{i}", pixel_num + 1, time,
     gray, normalized])
```

Listing 4.4: Extracting Engagement Data from PNG Heatmaps

4.2.5 Calculating Segment Popularity

The `calculate_segment_popularity()` function computes the average user engagement for predefined video segments using data extracted from the heatmap images. This data is crucial for identifying the most popular segments of a video, which can guide prefetching strategies in NDN.

This function calculates the average engagement for all the segments and then sorts them in descending order from highest average engagement to lowest.
The average engagement for a segment i is calculated as:

$$\text{Average Engagement}_i = \frac{\sum_{t \in S_i} E_t}{|S_i|}$$

where:

- S_i: The set of time points within the start and end time of segment i.
- E_t: The engagement percentage at time t.
- $|S_i|$: The number of time points in segment i.

```
def calculate_segment_popularity(input_csv, output_json):
    """
    Calculate and save segment popularity.

    Args:
        input_csv (str): Path to the CSV file containing engagement
    data.
        output_json (str): Path to the output JSON file for saving
   segment popularity.
```

```
    """
    # Load the engagement data CSV
    engagement_data = pd.read_csv(input_csv)

    # Calculate average engagement for each segment
    segment_popularity = []
    for i, (start, end) in enumerate(segments, start=1):
        start_seconds = int(start.total_seconds())
        end_seconds = int(end.total_seconds())

        # Filter data for the current segment
        segment_data = engagement_data[
            (engagement_data['Time (s)'] >= start_seconds) &
            (engagement_data['Time (s)'] < end_seconds)
        ]

        # Calculate average engagement
        average_engagement = segment_data['Normalized Engagement
    (%)'].mean()

        # Append to the segment popularity list
        segment_popularity.append({
            "Segment": i,
            "Start Time": str(start),
            "End Time": str(end),
            "Average Engagement": average_engagement
        })

    # Sort segments by average engagement
    sorted_segments = sorted(segment_popularity, key=lambda x: x['
    Average Engagement'], reverse=True)

    # Save to JSON file
    with open(output_json, "w") as f:
        json.dump(sorted_segments, f, indent=4)

    print(f"Segment popularity data saved to {output_json}")
```

Listing 4.5: Segment Popularity Calculation

Output

The JSON file generated by this function contains the following information for each segment:

- **Segment Number:** The index of the segment.
- **Start Time:** The starting time of the segment.
- **End Time:** The ending time of the segment.
- **Average Engagement:** The average engagement percentage for the segment.

This structured output enables further analysis and provides a direct input for the later exaplained prefetching algorithms. Here is an example of what the output-file may look like after running the pipeline:

```
{
    "Segment": 13,
    "Start Time": "0:35:20",
    "End Time": "0:39:00",
    "Average Engagement": 61.34027777777778
},
{
    "Segment": 12,
    "Start Time": "0:30:28",
    "End Time": "0:35:20",
    "Average Engagement": 59.34361924686193
},
```

Listing 4.6: Example Output data

After processing the extracted data, we can convert it into a readable format for analysis. For example, the engagement data for a specific video can be visualized as shown in Figure 4.3. This chart highlights viewer engagement over time, providing normalized percentages based on the extracted YouTube heatmap data.

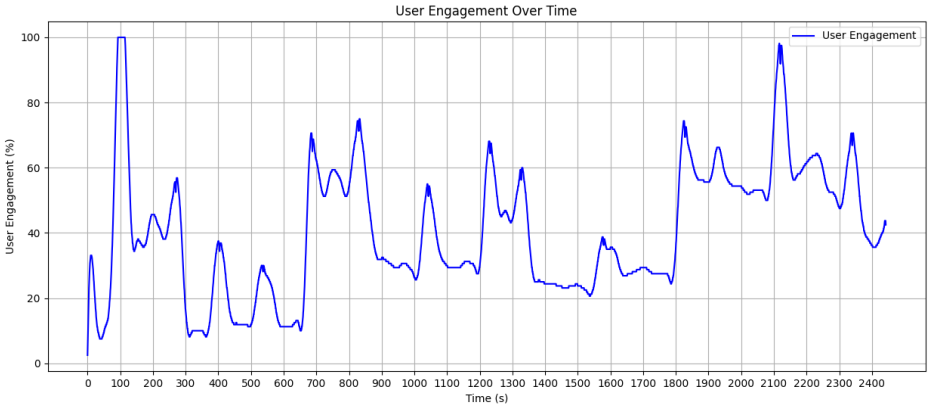

Figure 4.3: User engagement over time as extracted and visualized by the pipeline. This data is derived from YouTube heatmaps and normalized to show viewer engagement percentages.

The raw heatmap data, as illustrated in Figure 4.4, serves as the primary source for these calculations. This visualization showcases the engagement intensity across the video timeline, with peaks indicating sections of high user interest. By leveraging such data, our pipeline can accurately identify high-engagement segments to prioritize in the prefetching algorithm.

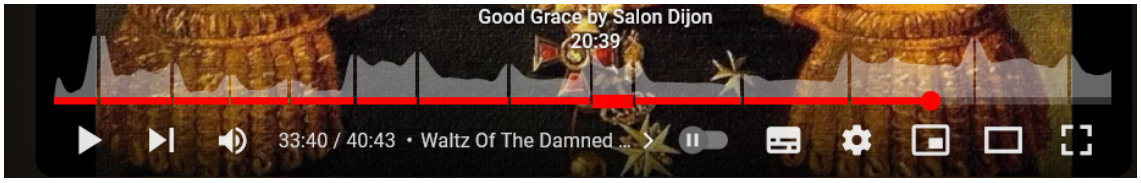

Figure 4.4: An example of a YouTube heatmap showing viewer engagement for a video. These heatmaps are the primary source of data for the prefetching pipeline.

4.3 Simulation Details

This section provides an in-depth description of the simulation framework and its design, including the network topology, simulation scenarios, and consumer behaviors implemented to evaluate the prefetching mechanism in NDN. The simulations were conducted using `ndnSIM` to emulate real-world content delivery dynamics. Each scenario highlights a specific aspect of content delivery, ranging from traditional CDN architectures to advanced prefetching mechanisms.

4.3.1 Network Topology

The network topology used in this study is based on the Abilene topology, consisting of 11 nodes distributed across major locations in the United States. These nodes represent routers, consumers, and producers. For Scenario 1, an alternate topology was created to mimic a traditional CDN architecture, where producers are located closer to the consumers.

Figure 4.1 illustrates the CDN-focused topology used in Scenario 1. Producers are located in Sunnyvale and Atlanta, ensuring low-latency delivery for consumers in nearby locations.

For Scenarios 2-4, the Abilene topology shown in Figure 4.2 was employed. In this setup, a single producer located in Houston serves as the content origin for all consumers, allowing for a comprehensive evaluation of prefetching and caching mechanisms under varied configurations.

Overall then we had to make one document detailing the topology, which is listed below. We could change the nodes from within the simulated scenarios to specify how the individual nodes should operate.

```
1
2  #name   city            latitude    longitude
3  ATLAng  Atlanta_GA      33.850000   -84.483300
4  CHINng  Chicago_IL      41.833300   -87.616700
5  DNVRng  Denver_CO       40.750000   -105.000000
6  HSTNng  Houston_TX      29.770031   -95.517364
7  IPLSng  Indianapolis_IN 39.780622   -86.159535
8  KSCYng  Kansas_City_MO  38.961694   -96.596704
9  LOSAng  Los_Angeles_CA  34.050000   -118.250000
10 NYCMng  New_York_NY     40.783300   -73.966700
11 SNVAng  Sunnyvale_CA    37.38575    -122.02553
12 STTLng  Seattle_WA      47.600000   -122.300000
13 WASHng  Washington_DC   38.897303   -77.026842
14
15 #x          y           capacity (kbps)
16 # Link connections
17 ATLAng      HSTNng      19000000
18 ATLAng      IPLSng      19000000
19 ATLAng      WASHng      19000000
20 ATLAng      KSCYng      19000000
21 CHINng      IPLSng      19000000
22 CHINng      NYCMng      19000000
23 DNVRng      KSCYng      19000000
24 DNVRng      SNVAng      19000000
25 DNVRng      STTLng      19000000
```

```
HSTNng      KSCYng      19000000
HSTNng      DNVRng      19000000
HSTNng      LOSAng      19000000
IPLSng      WASHng      19000000
IPLSng      CHINng      19000000
IPLSng      KSCYng      19000000
LOSAng      SNVAng      19000000
LOSAng      HSTNng      19000000
NYCMng      WASHng      19000000
SNVAng      STTLng      19000000
```

Listing 4.7: Base topology specifying nodes with their allotted coordinates, links and connections.

4.3.2 Simulation Scenarios

Four distinct simulation scenarios were designed to evaluate the performance of the proposed prefetching mechanism under different network configurations, consumer behaviors, and caching strategies. Each scenario incrementally introduces and tests caching, prefetching, and replacement mechanisms to assess their impact on content delivery performance.

Scenario 1: Traditional CDN Architecture

Scenario 1 simulates a traditional CDN architecture, where producers are located near the consumers to minimize latency. In this configuration, caching features are disabled, and all content retrieval is handled directly by the producers. This scenario serves as a baseline to compare the performance of NDN-based approaches.

The topology in this scenario uses a custom configuration designed to mimic a CDN, with producers located in Sunnyvale and Atlanta and consumers distributed in nearby locations. The nodes were configured using the following code shown in listing 4.8

```
CommandLine cmd;
std::string prefix1 = "IPLSng/Segmented_vid";
std::string prefix2 = "DNVRng/Segmented_vid";
cmd.Parse(argc, argv);
AnnotatedTopologyReader topologyReader("", 25);
topologyReader.SetFileName("src/ndnSIM/examples/topologies/topo-
 abilene-new.txt");
topologyReader.Read();

// Install NDN stack on all nodes
ndn::StackHelper ndnHelper;
//ndnHelper.setPolicy("nfd::cs::lru");
ndnHelper.setCsSize(0);
ndnHelper.InstallAll();

// Choosing forwarding strategy
ndn::StrategyChoiceHelper::InstallAll("/prefix", "/localhost/nfd/
 strategy/best-route");

```

```
18  // Installing global routing interface on all nodes
19  ndn::GlobalRoutingHelper ndnGlobalRoutingHelper;
20  ndnGlobalRoutingHelper.InstallAll();
21
22  //Initializing the consumers
23  Ptr<Node> consumers[4] = {Names::Find<Node>("NYCMng"), Names::
     Find<Node>("CHINng"), Names::Find<Node>("STTLng"), Names::Find<
     Node>("SNVAng")};
24
25  //Initializing the routers
26  Ptr<Node> routers[4] = {Names::Find<Node>("LOSAng"), Names::Find<
     Node>("ATLAng"), Names::Find<Node>("KSCYng"), Names::Find<Node
     >("WASHng")};
27
28  Ptr<Node> producer[2] = {Names::Find<Node>("IPLSng"), Names::Find
     <Node>("DNVRng")};
29
30  ndn::AppHelper producerHelper("ns3::ndn::Producer");
31  producerHelper.SetAttribute("PayloadSize", StringValue("1024"));
32
33  // Producer1 setup
34  producerHelper.SetPrefix(prefix1);
35  producerHelper.Install(producer[0]);
36
37  // Producer2 setup
38  producerHelper.SetPrefix(prefix2);
39  producerHelper.Install(producer[1]);
40
41  // Set up global routing for producer1
42  ndnGlobalRoutingHelper.AddOrigins(prefix1, producer[0]);
43
44  // Set up global routing for producer2
45  ndnGlobalRoutingHelper.AddOrigins(prefix2, producer[1]);
46
47  ndnGlobalRoutingHelper.CalculateRoutes();
```

Listing 4.8: Node setup for scenario 1

This scenario features two producers, each managing its own set of consumers to simulate a CDN-like setup. The nodes named IPLSng and DNVRng were designated as producers. Once the producers were configured, the *handleConsumerBehavior* function was used to define different behaviors for the consumers. After establishing consumer behavior, the simulation was set up to specify which producers the consumers would fetch content from. See listing **??**

```
1     std::vector<std::string> consumerBehaviors = {"SEQUENTIAL", "
     SKIPPER", "POPULAR_ONLY", "RANDOM"};
2     for (size_t i = 0; i < consumers.size(); i++) {
3         handleConsumerBehavior(consumers[i], consumerBehaviors[i],
     sequentialSegments, popularSegments, prefix, fetchStartTime,
     logFile);
4     }
5     for (int i = 0; i < 2; i++) {
6       consumerHelper.SetPrefix(segmentPrefix1);
7       ApplicationContainer app = consumerHelper.Install(consumers[i
     ]);
```

```
8       app.Start(Seconds(startTime));
9     }
10    for (int i = 2; i < 4; i++) {
11      consumerHelper.SetPrefix(segmentPrefix2);
12      ApplicationContainer app = consumerHelper.Install(consumers[i
      ]);
13      app.Start(Seconds(startTime));
14    }
```

Listing 4.9: Calling function to setup consumer behaviour and initialize producer prefix for the consumers. label

Scenario 2: NDN with Basic Caching

Scenario 2 evaluates NDN with standard caching mechanisms, such as Leave Copy Everywhere (LCE) and Least Recently Used (LRU). The producer is located in Houston, representing a cloud-based content server. This scenario does not employ prefetching, focusing instead on the impact of basic caching features in reducing latency and improving hit ratios. The main differences for the setup of this compared to scenario 1 is the setup of the producers and consumers, where we only have 1 producer *HSTNng* and we have to initialize simulation to use the replacement policy LRU and have a cache size bigger than 0. These differences have been illustrated in listing 4.10

```
1   ndn::StackHelper ndnHelper;
2   ndnHelper.setPolicy("nfd::cs::lru");
3   ndnHelper.setCsSize(500);
4   ndnHelper.InstallAll();
5
6   Ptr<Node> consumers[4] = {Names::Find<Node>("NYCMng"), Names::
      Find<Node>("CHINng"), Names::Find<Node>("STTLng"), Names::Find<
      Node>("SNVAng")};
7
8   Ptr<Node> routers[6] = {Names::Find<Node>("LOSAng"), Names::Find<
      Node>("DNVRng"), Names::Find<Node>("ATLAng"), Names::Find<Node
      >("KSCYng"), Names::Find<Node>("IPLSng"), Names::Find<Node>("
      WASHng")};
9
10  Ptr<Node> producer = Names::Find<Node>("HSTNng");
11
12  for (int i = 0; i < 4; i++) {
13    ndn::AppHelper consumerHelper("ns3::ndn::ConsumerCbr");
14    consumerHelper.SetAttribute("Frequency", StringValue("10")); //
      100 interests a second
15
16    // Each consumer will express the same data /root/<seq-no>
17    consumerHelper.SetPrefix("/HSTNng");
18    ApplicationContainer app = consumerHelper.Install(consumers[i])
      ;
19    app.Start(Seconds(0.01 * i));
20  }
21
```

```
22   ndnGlobalRoutingHelper.AddOrigins("/HSTNng", producer);
```

Listing 4.10: Scenario 2 differences regarding setup.

Here's a simplified version of the text, written in an easy-to-understand style:

Scenario 3: Prefetching Without Replacement Policies

In Scenario 3, we evaluate the impact of prefetching on content delivery performance without applying cache replacement policies like FIFO or LRU. The prefetching mechanism is implemented at the forwarding layer and integrates with the existing best route strategy. When a router receives an interest packet, it proactively begins fetching the most popular video segments associated with that interest. This means that while the router serves the initially requested data, it simultaneously prefetches additional high-engagement segments based on popularity metrics.

The implementation of the prefetching strategy is shown in Listing 4.11. The strategy initializes by loading a list of popular segments from a JSON file. Upon receiving an interest packet, it identifies the relevant video prefix and triggers prefetching for the associated popular segments. The process ensures that prefetched segments are fetched proactively and cached closer to the consumer.

```
1  NFD_LOG_INIT(PrefetchingStrategy);
2  NFD_REGISTER_STRATEGY(PrefetchingStrategy);
3
4  const Name& PrefetchingStrategy::getStrategyName() {
5    static const auto strategyName = Name("/localhost/nfd/strategy/
       prefetching").appendVersion(1);
6    //static const Name strategyName("/localhost/nfd/strategy/
       prefetching/1");
7    NFD_LOG_DEBUG("getStrategyName() returns: " << strategyName);
8    return strategyName;
9  }
10
11 PrefetchingStrategy::PrefetchingStrategy(Forwarder& forwarder,
       const Name& name)
12     : BestRouteStrategy(forwarder, name) {
13   ParsedInstanceName parsed = parseInstanceName(name);
14   if (!parsed.parameters.empty()) {
15     NDN_THROW(std::invalid_argument("PrefetchingStrategy does not
       accept parameters"));
16   }
17   if (parsed.version && *parsed.version != getStrategyName()[-1].
       toVersion()) {
18     NDN_THROW(std::invalid_argument(
19       "PrefetchingStrategy does not support version " + to_string(*
       parsed.version)));
20   }
21   this->setInstanceName(makeInstanceName(name, getStrategyName()));
22   NFD_LOG_DEBUG("PrefetchingStrategy initialized with instance name
       : " << this->getInstanceName());
23   loadPopularSegments("/path/to/json");
24 }
25
```

```
26 void PrefetchingStrategy::loadPopularSegments(const std::string&
     jsonFilePath) {
27   std::ifstream file(jsonFilePath);
28   if (!file.is_open()) {
29     NFD_LOG_ERROR("Unable to open JSON file for popular segments");
30     return;
31   }
32
33   nlohmann::json data;
34   file >> data;
35
36   for (const auto& segment : data) {
37     popularSegments.push_back(segment["Segment"]);
38   }
39 }
40
41 void PrefetchingStrategy::afterReceiveInterest(const ::ndn::
     Interest& interest,
42                                                const FaceEndpoint&
     ingress,
43                                                const std::
     shared_ptr<pit::Entry>& pitEntry) {
44   BestRouteStrategy::afterReceiveInterest(interest, ingress,
     pitEntry);
45
46   const ::ndn::Name& currentInterest = interest.getName();
47   ::ndn::Name videoPrefix = currentInterest.getPrefix(-1);
48
49   for (const auto& segmentId : popularSegments) {
50     ::ndn::Name prefetchSegment = videoPrefix;
51     prefetchSegment.append("segment_" + std::to_string(segmentId));
52     NFD_LOG_DEBUG("Prefetching segment: " << prefetchSegment);
53     triggerPrefetch(prefetchSegment, pitEntry);
54   }
55 }
56
57 void PrefetchingStrategy::triggerPrefetch(const ::ndn::Name&
     segmentName,
58                                           const std::shared_ptr<pit
     ::Entry>& pitEntry) {
59   const fib::Entry& fibEntry = this->lookupFib(*pitEntry);
60   const fib::NextHopList& nexthops = fibEntry.getNextHops();
61
62   ::ndn::Interest prefetchInterest(segmentName);
63   prefetchInterest.setCanBePrefix(false);
64   prefetchInterest.setInterestLifetime(::ndn::time::seconds(1));
65
66   for (const auto& nexthop : nexthops) {
67     const nfd::face::Face& face = nexthop.getFace();
68     if (isNextHopEligible(face, prefetchInterest, nexthop, pitEntry
     )) {
69       this->sendInterest(prefetchInterest, const_cast<nfd::face::
     Face&>(face), pitEntry);
70       NFD_LOG_DEBUG("Prefetch triggered for segment: " <<
     segmentName << " via Face: " << face.getId());
71       return;
72     }
```

```
73   }
74
75   NFD_LOG_DEBUG("No eligible next-hop for prefetching: " <<
      segmentName);
76 }
```

Listing 4.11: Prefetching implementation

The prefetching strategy is activated during the setup phase of the simulation by assigning it as the default strategy using the *StrategyChoiceHelper*. Listing 4.12 demonstrates how this is configured in the simulation. The simulation uses a JSON file containing the popularity metrics to identify high-engagement segments for prefetching.

```
1     ndn::StackHelper ndnHelper;
2     //ndnHelper.setPolicy("nfd::cs::lru"); outcommented for
     scenario 3
3
4     ndn::StrategyChoiceHelper::InstallAll("/", "/localhost/nfd/
     strategy/PrefetchingStrategy");
5
6     std::string prefix = "/Video";
7     std::string jsonFilePath = "/segment_popularity.json";
8     cmd.AddValue("jsonFile", "Path to the JSON file containing
     segment popularity data", jsonFilePath);
9     cmd.Parse(argc, argv);
10
11    // Load segments
12    std::vector<Segment> sequentialSegments, popularSegments;
13    const float engagementThreshold = 50.0f;
14    loadSegments(jsonFilePath, sequentialSegments, popularSegments,
      engagementThreshold);
```

Listing 4.12: Scenario 3 setup.

Scenario 4: Prefetching with Replacement Policies

Scenario 4 incorporates the prefetching mechanism alongside caching policies such as FIFO and LRU. This scenario aims to simulate a realistic environment where the cache replacement strategy complements prefetching. As in Scenario 3, multiple cache sizes were tested to evaluate the trade-offs between cache size, hit ratios, and network efficiency. This scenario is the same as scenario 3, but with replacement policies enabled and alternated between depending on the test.

All these scenarios utilize the same form of consumer behavior methods, which will be explained in the next subsection.

4.3.3 Consumer Behavior Models

To simulate diverse user interactions with video content, four distinct consumer behaviors were implemented. Each behavior reflects a unique pattern of content requests, allowing for a detailed evaluation of the prefetching mechanism's adaptability:

Sequential Behavior: Consumers request video segments in order, starting from the beginning. This behavior represents a typical video streaming scenario.

Skipper Behavior: Consumers skip alternate segments, simulating users who frequently jump through video content.

Popular-Only Behavior: Consumers request only the most popular segments of a video, based on predefined popularity metrics.

Random Behavior: Consumers request video segments in a random order, representing unpredictable access patterns.

The function `handleConsumerBehavior()` was used to define and manage these behaviors. Each behavior type determines the segments to fetch, ensuring the simulation reflects real-world variability in user interactions with video content.

```
void handleConsumerBehavior(Ptr<Node> consumer, const std::string&
    behaviorType,
                            const std::vector<Segment>&
    sequentialSegments,
                            const std::vector<Segment>&
    popularSegments,
                            const std::string& prefix,
                            double& currentTime, std::ofstream&
    logFile) {

     std::vector<Segment> segmentsToFetch;

     if (behaviorType == "SEQUENTIAL") {
         segmentsToFetch = sequentialSegments;
     } else if (behaviorType == "SKIPPER") {
         for (size_t i = 0; i < sequentialSegments.size(); i += 2) {
             segmentsToFetch.push_back(sequentialSegments[i]);
         }
     } else if (behaviorType == "POPULAR_ONLY") {
         segmentsToFetch = popularSegments;
     } else if (behaviorType == "RANDOM") {
         segmentsToFetch = sequentialSegments;
         std::random_shuffle(segmentsToFetch.begin(),
    segmentsToFetch.end());
     } else {
         logFile << "ERROR: Unknown behavior type: " << behaviorType
     << "\n";
         return;
     }

     for (const auto& segment : segmentsToFetch) {
         std::string fetchPrefix = prefix + "/segment_" + std::
    to_string(segment.id) + ".mp4";

         if (!isSegmentCached(consumer, fetchPrefix)) {
             ndn::AppHelper consumerHelper("ns3::ndn::ConsumerCbr");
             consumerHelper.SetAttribute("Frequency", StringValue("
    10"));
             consumerHelper.SetPrefix(fetchPrefix);

             ApplicationContainer fetchApp = consumerHelper.Install(
    consumer);
```

```
34              fetchApp.Start(Seconds(currentTime));
35
36              logFile << "FETCH (" << behaviorType << "): Consumer: "
     << consumer->GetId()
37                      << ", Segment: " << fetchPrefix
38                      << ", Time: " << std::fixed << std::
    setprecision(2) << currentTime << " seconds\n";
39
40              currentTime += 1.00;
41          }
42      }
43  }
```

Listing 4.13: Consumer Behavior Simulation Code

While the focus of this section is on the scenarios and behaviors, it is important to note that Scenarios 2, 3, and 4 were tested with cache sizes of 500, 1000, and 1500 entries. The impact of these sizes on performance will be detailed in the chapter Testing and Results 5. Similarly, replacement policies like FIFO and LRU were explored to complement the prefetching mechanism in Scenario 4.

Chapter 5

Testing and results

5.1 Evaluation Metrics

To evaluate the performance of the proposed prefetching mechanism, several metrics were calculated, including Average Delay, Cache Hit Ratio (CHR), Path Reduction, and Retransmissions. These metrics offer insights into the efficiency and reliability of the prefetching algorithm under different scenarios.

Average Delay The average delay is calculated as the mean delay for each node and request type:

$$\text{Average Delay}_{n,t} = \frac{\sum_{i=1}^{N} \text{Delay}_{n,t,i}}{N}$$

Cache Hit Ratio (CHR) The Cache Hit Ratio is given by:

$$\text{CHR} = \frac{\text{Total Cache Hits}}{\text{Total Cache Hits} + \text{Total Cache Misses}} \times 100$$

Path Reduction Path Reduction measures the reduction in hop count relative to a predefined full path length:

$$\text{Path Reduction} = 1 - \frac{\text{Hop Count}}{\text{Full Path Length}}$$

Hop Count The average hop count per node is calculated as:

$$\text{Average Hop Count}_n = \frac{\sum_{i=1}^{N} \text{Hop Count}_{n,i}}{N}$$

Retransmissions The total number of retransmissions per node is:

$$\text{Total Retransmissions}_n = \sum_{i=1}^{N} \text{RetxCount}_{n,i}$$

5.2 Implementation of Custom Cache Hit Measurement

One of the significant challenges encountered during testing was the absence of built-in cache hit tracking in `ndnSIM 3.35`. To address this, a custom mechanism was implemented to log and track cache hits during simulations. This implementation ensured accurate measurement of the cache hit ratio across all scenarios.

The custom implementation involved modifying the content store functionality and the cs-tracer script which is supposed to track these events to log every cache lookup and its result. Below is a snippet of the code that was added to track cache hits and misses:

```
// remade this function for ndnsim version 3.35 by using a roundabout method through L3Protocol
void
CsTracer::Connect()
{
  // Access L3Protocol from the node
  auto l3Protocol = m_nodePtr->GetObject<ndn::L3Protocol>();
  if (!l3Protocol) {
    NS_LOG_WARN("L3Protocol not found on node " << m_nodePtr->GetId());
    return;
  }

  // Access Forwarder through L3Protocol
  auto forwarderPtr = l3Protocol->getForwarder();
  if (!forwarderPtr) {
    NS_LOG_WARN("Forwarder not found on node " << m_nodePtr->GetId());
    return;
  }

  // Dereference to access the Forwarder
  auto& forwarder = *forwarderPtr;

  // Access Content Store
  auto& cs = forwarder.getCs();

  // Connect signals, Potentially also make this such that it spits out the failed interest and the data from the fetched cahce
  cs.cacheHit.connect([this](const ndn::Data& data) {
    this->CacheHits(data);
  });

  cs.cacheMiss.connect([this](const ndn::Interest& interest) {
    this->CacheMisses(interest);
  });

  NS_LOG_DEBUG("CsTracer connected to Content Store on node " << m_nodePtr->GetId());
  Reset();
}

```

```
38 //New methods just changed the data types
39 void CsTracer::CacheHits(const ndn::Data& data)
40 {
41   m_stats.m_cacheHits++;
42 }
43
44 void CsTracer::CacheMisses(const ndn::Interest& interest)
45 {
46   m_stats.m_cacheMisses++;
47 }
```

Listing 5.1: Modified CsTracer connect function to make a proper call when incrementing CacheHits and Cahcemisses.

It calls the signal protocol and counts the cachehits and cachemisses based on the following which gets called in cs.cpp:

```
1  // Define signals for cache events
2    boost::signals2::signal<void(const Data&)> cacheHit;       //
       Fired on cache hit
3    boost::signals2::signal<void(const Interest&)> cacheMiss;  //
       Fired on cache miss
4
5  Cs::const_iterator
6  Cs::findImpl(const Interest& interest) const
7  {
8    if (!m_shouldServe || m_policy->getLimit() == 0) {
9      return m_table.end();
10   }
11
12   const Name& prefix = interest.getName();
13   auto range = findPrefixRange(prefix);
14   auto match = std::find_if(range.first, range.second,
15                             [&interest] (const auto& entry) {
       return entry.canSatisfy(interest); });
16
17   if (match == range.second) {
18     NFD_LOG_DEBUG("find " << prefix << " no-match");
19     // Emit cache miss signal
20     cacheMiss(interest);
21     return m_table.end();
22   }
23   NFD_LOG_DEBUG("find " << prefix << " matching " << match->getName
       ());
24
25   // Extract ndn::Data from nfd::cs::Entry and emit it as cache hit
       signal
26   cacheHit(match->getData());
27   m_policy->beforeUse(match);
28
29   return match;
30 }
```

Listing 5.2: Changes to cs.cpp, to make cs-tracer.cpp able to handle the cachehit and cachemiss events properly. The main thing done here is the definition of the signals cache hit and miss.

With those implementations done then we could start measuring the cache hits and cache misses, thereby calculate the cache hit ratios.

5.3 Results

This section details the results aquired after thorough testing for each scenario the graphs and results themselves are however moved to Appendix A.

5.3.1 Cache Hit Ratio (CHR) results

The overall CHR was calculated for scenarios 2-4 across different cache sizes. The results indicate the effectiveness of the caching mechanism in reducing content retrieval latency. The graphs are located in Appendix A Section A.1

5.3.2 Average Delay

The average delay (D_{avg}) was calculated for each node and plotted for each scenario. This provides a clear view of delay trends across the network. The graphs are located in Appendix A Section A.2

5.3.3 Path Reduction

Path reduction was measured to evaluate the impact of caching on reducing the average hop count. The results for the different scenarios are shown below. The graphs are located in Appendix A Section A.3

5.3.4 Retransmissions

Retransmission statistics provide insights into the reliability of the network. The results for each scenario are presented below. The graphs are located in Appendix A Section A.4

5.3.5 Hop Count

The average hop count per node was calculated to understand how caching impacts routing efficiency. The graphs are located in Appendix A Section A.5

Chapter 6

Discussion

The evaluation of Named Data Networking (NDN) scenarios reveals significant insights into the efficiency of different caching and prefetching strategies, based on the analysis of delay metrics, hop counts, retransmissions, and cache hit ratio (CHR). Each scenario demonstrates unique strengths and weaknesses, contributing to an overall understanding of network performance optimization.

Scenario 1, characterized by a CDN-inspired topology, exhibited the lowest delays and hop counts across all metrics. The extitFullDelay and extitLastDelay metrics were consistently low, averaging around 622 μs for most nodes. Similarly, the average hop count for all consumers was 1.0, indicating that content was effectively cached near consumer nodes. However, the retransmission count of 5312 for all consumers revealed inefficiencies in caching mechanisms, suggesting that this topology, while optimized for delay and proximity, lacked advanced caching strategies to reduce redundancy.

Scenario 2 employed simple NDN with replacement policies, achieving significant reductions in retransmissions. Consumers like *CHINng*, *SNVAng*, and *STTLng* experienced retransmission counts as low as 400, while *NYCMng* recorded 399. This scenario also demonstrated the highest overall CHR, maintaining a constant 24.56% across all Content Store (CS) sizes. However, Scenario 2 suffered from high delays, with extitNYCMng showing extitFullDelay values exceeding 2400 μs, and consistently higher hop counts of 3.0 for most consumer nodes. These results indicate that while replacement policies optimize cache utilization, they fail to minimize delays effectively.

Scenario 3 introduced prefetching mechanisms without replacement policies, resulting in mixed outcomes. Delays were moderate compared to Scenario 2, with consumer nodes like *NYCMng* and *WASHng* exhibiting extitFullDelay values ranging from 1725 μs to 1798 μs. Hop counts remained steady at 2.0 for most consumers, while retransmissions increased significantly for nodes such as *NYCMng* (5312) and *SNVAng* (5246). The overall CHR values ranged from 21.52% at 500CS to 24.08% at 1500CS, reflecting a less efficient cache utilization compared to Scenario 2. These results highlight the limitations of prefetching as a standalone mechanism, particularly in reducing retransmissions and optimizing CHR.

Scenario 4 combined prefetching with replacement policies, yielding balanced performance across all metrics. Delays were moderate, with extitCHINng maintaining stable extitFullDelay and extitLastDelay values of approximately 1059 μs.

Hop counts for consumers like *SNVAng* and *STTLng* remained consistent at 2.0, and retransmissions matched those observed in Scenario 3. The CHR values in Scenario 4 demonstrated improvements over Scenario 3, particularly in the FIFO configuration, achieving 21.69% at 500CS and converging with LRU at 24.08% at 1500CS. These results emphasize the complementary role of replacement policies in enhancing prefetching strategies and achieving efficient caching.

In summary, Scenario 1 excelled in minimizing delays and hop counts but had high retransmissions. Scenario 2 achieved the highest CHR and minimized retransmissions through replacement policies but suffered from high delays and traversal distances. Scenario 3 demonstrated the potential of prefetching mechanisms but was limited by the absence of replacement policies, leading to higher retransmissions and suboptimal CHR. Scenario 4 achieved a balanced performance by combining prefetching with replacement policies, addressing the limitations observed in Scenario 3 while maintaining moderate delays and hop counts. These findings underscore the importance of integrating the prefetching strategy with a robust caching mechanisms to optimize NDN performance effectively.

6.1 Future Work and Improvements

This section discusses the primary improvements and future work that need to be made, for the prefetching module to become better, it should focus on the following key areas.

6.1.1 Dynamic Prefetching Strategies

The prefetching mechanism should adapt to real-time traffic patterns and user behavior. By employing machine learning models to predict content popularity dynamically, the system can make more informed caching decisions, minimizing unnecessary prefetching and improving overall cache hit ratios.

6.1.2 Hybrid Replacement Policies

Combining prefetching with advanced hybrid replacement policies, such as integrating LRU and LFU, can balance short-term and long-term caching needs. This approach will enhance content availability while optimizing the utilization of Content Store resources.

6.1.3 Consumer-Centric Optimization

Focusing prefetching efforts on high-demand consumer nodes, such as *NYCMng* and *SNVAng*, can significantly reduce delays and retransmissions. Prioritizing content for these nodes will ensure better network efficiency and user experience.

6.1.4 Proactive Cache Management

Implementing proactive replacement strategies based on predicted future requests and content popularity can optimize cache performance. By replacing less popular

or outdated content preemptively, the module can maintain higher hit ratios and minimize redundant requests.

By implementing these targeted improvements, the prefetching module can enhance its efficiency, reduce network delays and retransmissions, and improve cache utilization. These advancements will contribute to the scalability and performance of NDN, making it a more robust and effective solution for modern networking challenges.

Chapter 7

Conclusive remarks

This chapter concludes on the hypothesis and purpose of the project, presented in section 1.2 and the project goals from section 1.3.

The project successfully explored the potential of Named Data Networking (NDN) combined with popularity-based prefetching to optimize video streaming performance. By evaluating key metrics such as average delay, hop count, retransmissions, path reduction, and cache hit ratio, the findings provided valuable insights into the strengths and limitations of various configurations. Scenario 1 demonstrated low-latency benefits inherent in CDN-inspired architectures but was hindered by high retransmissions due to limited caching mechanisms. Scenario 2 effectively leveraged replacement policies to enhance cache utilization and reduce retransmissions, though it exhibited higher delays. Scenario 3 highlighted the potential of prefetching to lower delays, but its lack of replacement policies resulted in higher retransmissions and suboptimal cache hit efficiency. Scenario 4, combining prefetching with caching strategies like FIFO and LRU, achieved a well-balanced performance across all metrics, demonstrating the synergistic benefits of integrating these approaches.

The primary goals of this project were to analyze the effectiveness of prefetching strategies and to identify their potential to enhance caching efficiency and reduce network overhead. These goals were largely met, as the results validated the benefits of prefetching when combined with robust caching mechanisms. However, the findings also revealed areas for improvement, particularly in optimizing retransmissions and fully leveraging cache sizes.

This work highlights the importance of adaptive and predictive strategies in prefetching to address the dynamic nature of video streaming demands. Future enhancements, such as machine learning-driven popularity predictions, hybrid caching policies, and network-aware optimizations, hold significant promise for further improving the scalability and efficiency of NDN architectures. This project represents a meaningful step toward realizing robust, user-centric content delivery solutions in modern networking environments.

Bibliography

[1] D. D. Clark, *Designing an Internet*. MIT Press, 2018, ISBN: 9780262535205. DOI: 10.7551/mitpress/11373.001.0001. [Online]. Available: https://doi.org/10.7551/mitpress/11373.001.0001.

[2] L. Zhang, A. Afanasyev, J. Burke, *et al.*, "Named data networking," *ACM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014. [Online]. Available: https://named-data.net/wp-content/uploads/2014/10/named_data_networking_ccr.pdf.

[3] G. Rossini and D. Rossi, "Coupling caching and forwarding: Benefits, analysis, and implementation," ser. ACM-ICN '14, Paris, France: Association for Computing Machinery, 2014, 127–136, ISBN: 9781450332064. DOI: 10.1145/2660129.2660153. [Online]. Available: https://doi.org/10.1145/2660129.2660153.

[4] I. Psaras, W. K. Chai, and G. Pavlou, "In-network cache management and resource allocation for information-centric networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 11, pp. 2920–2931, 2014. DOI: 10.1109/TPDS.2013.304.

[5] G. Carofiglio, M. Gallo, and L. Muscariello, "Joint hop-by-hop and receiver-driven interest control protocol for content-centric networks," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, 491–496, Sep. 2012, ISSN: 0146-4833. DOI: 10.1145/2377677.2377772. [Online]. Available: https://doi.org/10.1145/2377677.2377772.

[6] W. Li, S. M. Oteafy, and H. S. Hassanein, "Dynamic adaptive streaming over popularity-driven caching in information-centric networks," in *2015 IEEE International Conference on Communications (ICC)*, 2015, pp. 5747–5752. DOI: 10.1109/ICC.2015.7249238.

[7] W. Li, Y. Li, W. Wang, Y. Xin, and T. Lin, "A popularity-driven caching scheme with dynamic multipath routing in ccn," in *2016 IEEE Symposium on Computers and Communication (ISCC)*, 2016, pp. 633–638. DOI: 10.1109/ISCC.2016.7543808.

[8] J. Postel, *Transmission control protocol*, RFC 793, 1981. [Online]. Available: https://www.rfc-editor.org/rfc/rfc793.

[9] R. Rejaie, M. Handley, and D. Estrin, "Layered quality adaptation for internet video streaming," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 12, pp. 2530–2543, 2000. DOI: 10.1109/49.898735.

[10] D. Wu, Y. Hou, W. Zhu, Y.-Q. Zhang, and J. Peha, "Streaming video over the internet: Approaches and directions," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, no. 3, pp. 282–300, 2001. DOI: 10.1109/76.911156.

[11] V. Jacobson, "Congestion avoidance and control," *Proceedings of ACM SIGCOMM*, pp. 314–329, 1988.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, *Hypertext transfer protocol – http/1.1*, https://tools.ietf.org/html/rfc2616, Accessed: 2024-09-15.

[13] G. Xylomenos, G. C. Polyzos, P. Mahonen, and M. Saaranen, "TCP performance issues over wireless links," *Communications Magazine*, vol. 39, no. 4, pp. 52–58, Apr. 2001, ISSN: 0163-6804. DOI: 10.1109/35.917504. [Online]. Available: https://doi.org/10.1109/35.917504.

[14] A. Nesterkin and V. Deart, "TCP is bottleneck of video streaming via OTT," in *Proceedings of the 24th Conference of Open Innovations Association FRUCT (FRUCT'24)*, Moscow, Russia: FRUCT Oy, 2019.

[15] ResellerClub Blog, *What is a cdn (content delivery network) and how does a cdn work?* Accessed: 2024-11-01. [Online]. Available: https://blog.resellerclub.com/what-is-cdn-content-delivery-network-how-does-cdn-work/.

[16] C. Wang, A. Jayaseelan, and H. Kim, "Comparing cloud content delivery networks for adaptive video streaming," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, 2018, pp. 686–693. DOI: 10.1109/CLOUD.2018.00094.

[17] B. Krishnamurthy, C. Wills, and Y. Zhang, "On the use and performance of content distribution networks," in *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement (IMW '01)*, San Francisco, California, USA: Association for Computing Machinery, 2001, pp. 169–182, ISBN: 1581134355. DOI: 10.1145/505202.505224. [Online]. Available: https://doi.org/10.1145/505202.505224.

[18] T. Stockhammer, "Dynamic adaptive streaming over HTTP – standards and design principles," in *Proceedings of the 2nd Annual ACM Conference on Multimedia Systems (MMSys)*, ACM, 2011, pp. 133–144. DOI: 10.1145/1943552.1943572. [Online]. Available: https://dl.acm.org/doi/10.1145/1943552.1943572.

[19] A. Langley, A. Riddoch, A. Wilk, *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ACM, 2017, pp. 183–196.

[20] Gumlet, *Tcp vs udp: Differences and comparison for video streaming*, Available at https://www.gumlet.com/learn/tcp-vs-udp/, 2023.

[21] G. Xylomenos, C. Ververidis, V. A. Siris, *et al.*, "A survey of information-centric networking research," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 1024–1049, 2014. [Online]. Available: https://ieeexplore.ieee.org/document/6563278.

[22] V. Jacobson, D. K. Smetters, J. D. Thornton, and et al., "Networking named content," in *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, ACM, 2009, pp. 1–12.

[23] L. Zhang, D. Estrin, J. Burke, *et al.*, "Named data networking (ndn) project," NDN Consortium, Tech. Rep., 2010. [Online]. Available: https://named-data.net/wp-content/uploads/TR001ndn-proj.pdf.

[24] A. Afanasyev, J. Shi, E. Uzun, and L. Zhang, "Ndns: A dns-like name service for ndn," in *2013 22nd International Conference on Computer Communications and Networks (ICCCN)*, IEEE, 2013, pp. 1–9.

[25] G. Carofiglio, M. Gallo, L. Muscariello, J.-L. Rougier, and J. Roberts, "From content delivery today to information centric networking," *Computer Communications*, vol. 37, pp. 36–49, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S1389128613002156.

[26] P. Gasti, G. Tsudik, E. Uzun, and L. Zhang, "DoS and DDoS in named data networking," in *2013 22nd International Conference on Computer Communication and Networks (ICCCN)*, IEEE, 2013, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/6614127.

[27] L. Zhang, A. Afanasyev, J. Burke, and et al., "Named data networking (ndn) project 2013-2014 annual report," NDN Technical Report NDN-0021, Tech. Rep., 2014.

[28] E. Rescorla, *HTTP Over TLS*, RFC 2818, 2000. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2818.

[29] Y. Yu, A. Afanasyev, B. Clark, and et al., "Schematizing trust in named data networking," *ACM ICN*, 2015.

[30] Z. Zhang, A. Afanasyev, Z. Zhu, and L. Zhang, "Public key management in named data networking," NDN Project, Technical Report NDN-0029, 2015. [Online]. Available: https://named-data.net/wp-content/uploads/2015/04/ndn-0029-1-public-key-management-ndn.pdf.

[31] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Computing Surveys*, vol. 35, no. 4, pp. 374–398, 2003. DOI: 10.1145/954339.954341.

[32] H. Dai, J. Wang, Z. Fan, B. Liu, and Y. Chen, "On pending interest table in named data networking," in *2017 International Conference on Computer Communication and Networks (ICCCN)*, IEEE, 2017, pp. 1–9. [Online]. Available: https://ieeexplore.ieee.org/document/7846700.

[33] A. Afanasyev, J. Shi, B. Zhang, and L. Zhang, "Interest flooding attack and countermeasures in named data networking," in *IFIP Networking Conference*, IEEE, 2013, pp. 1–9.

[34] M. Zhang, X. Wang, T. Liu, *et al.*, "AFSndn: A novel adaptive forwarding strategy in named data networking based on Q-learning," *Peer-to-Peer Networking and Applications*, vol. 13, no. 4, pp. 1176–1184, 2020. DOI: 10.1007/s12083-019-00845-w. [Online]. Available: https://doi.org/10.1007/s12083-019-00845-w.

[35] O. Akinwande and E. Gelenbe, "A Reinforcement Learning Approach to Adaptive Forwarding in Named Data Networking," in *Computer and Information Sciences. ISCIS 2018. Communications in Computer and Information Science*, T. Czachórski, E. Gelenbe, K. Grochla, and R. Lent, Eds., vol. 935, Springer, Cham, 2018, pp. 281–292. DOI: 10.1007/978-3-030-00840-6_23. [Online]. Available: https://doi.org/10.1007/978-3-030-00840-6_23.

[36] W. Ali, S. M. H. Shamsuddin, and A. S. Ismail, "A survey of web caching and prefetching," 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:15831431.

[37] M. A. Hoque, M. M. Hasan, and M. S. Islam, "Caching on named data network: A survey and future research," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 8, no. 5, pp. 2995–3007, 2018.

[38] A. Afanasyev, J. Burke, and L. Zhang, "The ndn testbed," NDN Technical Report NDN-0005, Tech. Rep., 2016.

[39] L. Zhang *et al.*, "Named data networking (ndn): An overview," *IEEE Communications Magazine*, vol. 50, no. 1, pp. 20–27, 2018. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/named-data-network.

[40] J. B. Schafer *et al.*, "Collaborative filtering for web recommendations," *IEEE Internet Computing*, vol. 5, no. 6, pp. 27–33, 2019.

[41] A. Detti, N. Blefari-Melazzi, S. Salsano, and M. Pomposini, "CONET: A content centric inter-networking architecture," in *Proceedings of the ACM SIGCOMM Workshop on Information-Centric Networking (ICN)*, 2011, pp. 50–55. [Online]. Available: https://conferences.sigcomm.org/sigcomm/2011/papers/icn/p50.pdf.

[42] H. Wang, L. Zhang, and Y. Li, "Routing in named data networking (technical report)," NDN Project, Tech. Rep., 2016, Accessed: 2024-09-15. [Online]. Available: https://named-data.net/wp-content/uploads/2013/12/TR16-routing.pdf.

[43] A. Afanasyev, I. Moiseenko, and L. Zhang, "Ndn security overview (ndn technical report ndn-0057)," Named Data Networking Project, Tech. Rep., 2018, Accessed: 2024-09-15. [Online]. Available: https://named-data.net/wp-content/uploads/2018/07/ndn-0057-4-ndn-security.pdf.

[44] Akamai, *What is a cdn?* https://www.akamai.com/glossary/what-is-a-cdn, Accessed: 2024-09-20.

[45] Y. Zhang, X. Tan, and W. Li, "In-network cache size allocation for video streaming on named data networking," in *Proceedings of the 2017 VI International Conference on Network, Communication and Computing (ICNCC '17)*, Kunming, China: Association for Computing Machinery, 2017, pp. 18–23, ISBN: 9781450353663. DOI: 10.1145/3171592.3171604. [Online]. Available: https://doi.org/10.1145/3171592.3171604.

[46] C. Fan, S. Shannigrahi, C. Papadopoulos, and C. Partridge, "Discovering in-network caching policies in ndn networks from a measurement perspective," in *Proceedings of the 7th ACM Conference on Information-Centric Networking (ICN '20)*, Virtual Event, Canada: Association for Computing Machinery, 2020, pp. 106–116, ISBN: 9781450380409. DOI: 10.1145/3405656.3418711. [Online]. Available: https://doi.org/10.1145/3405656.3418711.

Appendices

Appendix A

Results

This appendix includes all the graphs made for each test case and each scenario, starting with the results related to Cache Hit ratio and ending with hop count.

A.1 Cache Hit Ratio (CHR)

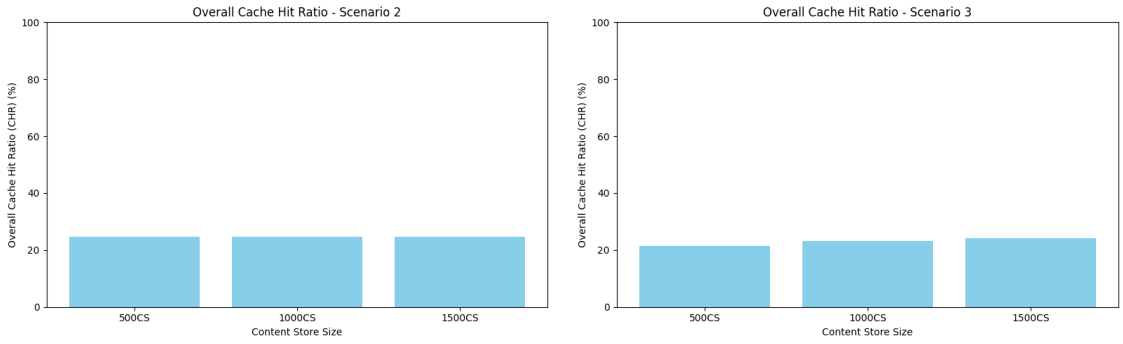

Figure A.1: Overall Cache Hit Ratio (CHR) for Scenario 2 on the left and Scenario 3 on the right.

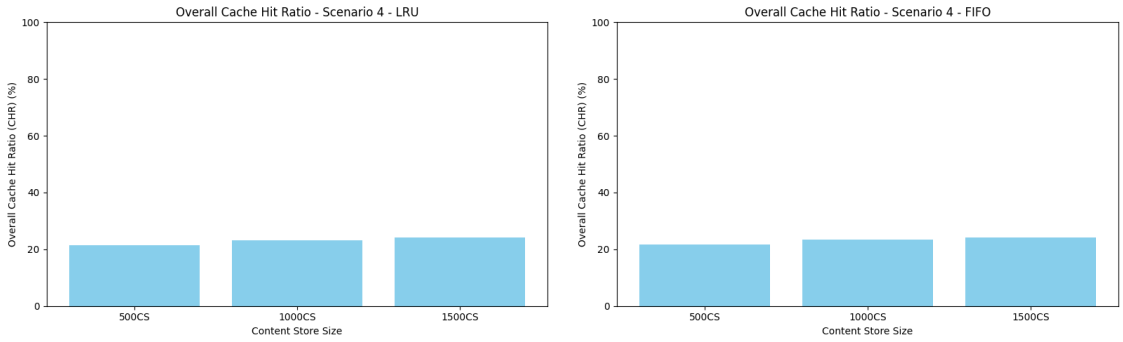

Figure A.2: Overall Cache Hit Ratio (CHR) for Scenario 4 with LRU caching on the left and FIFO on the right.

A.2 Average Delay

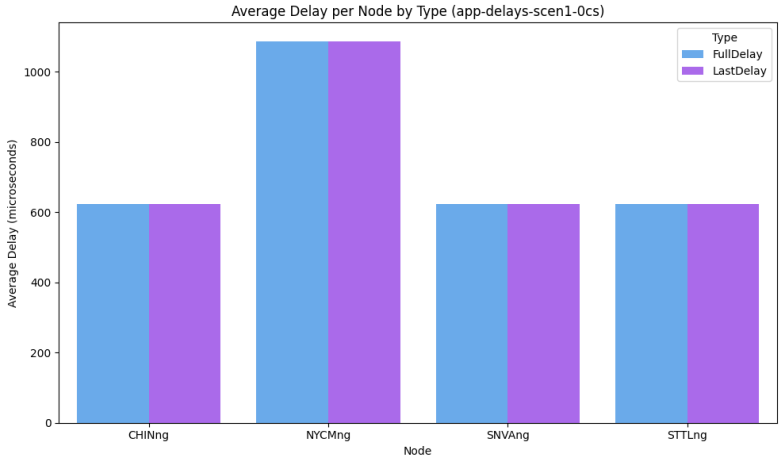

Figure A.3: Average delay per node for Scenario 1.

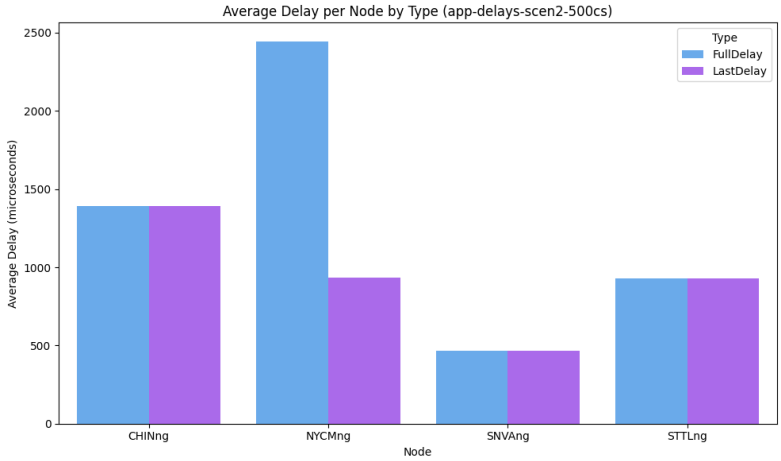

Figure A.4: Average delay per node for Scenario 2 with 500 CS.

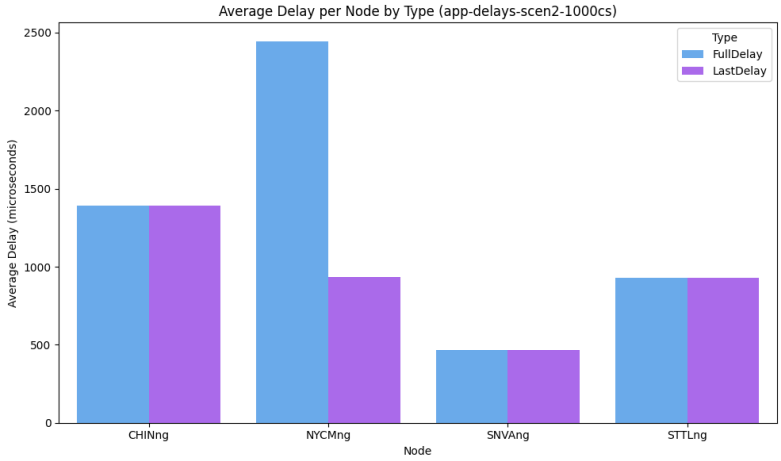

Figure A.5: Average delay per node for Scenario 2 with 1000 CS.

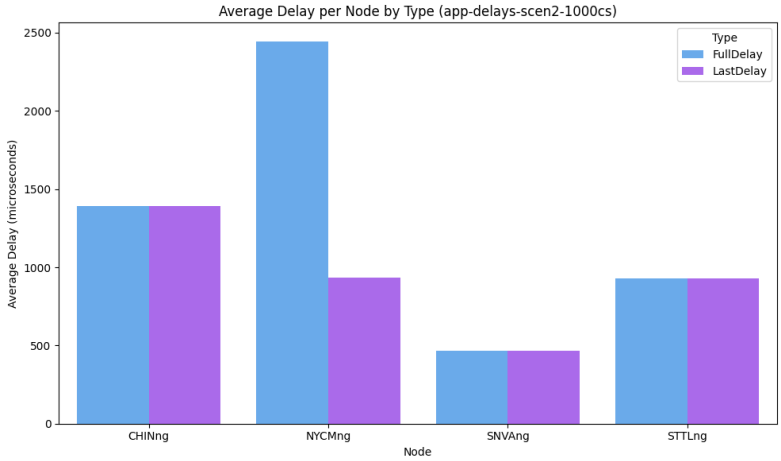

Figure A.6: Average delay per node for Scenario 2 with 1500 CS.

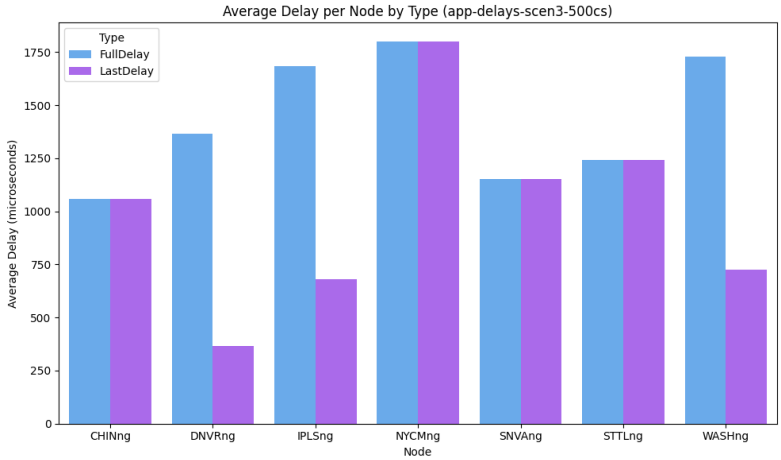

Figure A.7: Average delay per node for Scenario 3 with 500 CS.

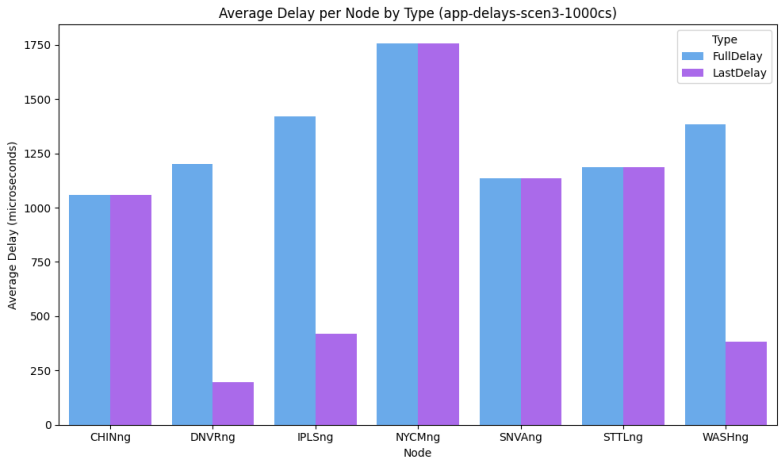

Figure A.8: Average delay per node for Scenario 3 with 1000 CS.

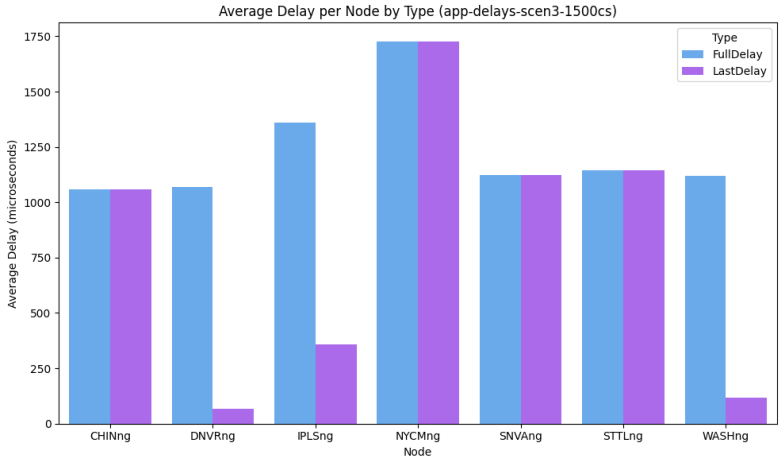

Figure A.9: Average delay per node for Scenario 3 with 1500 CS.

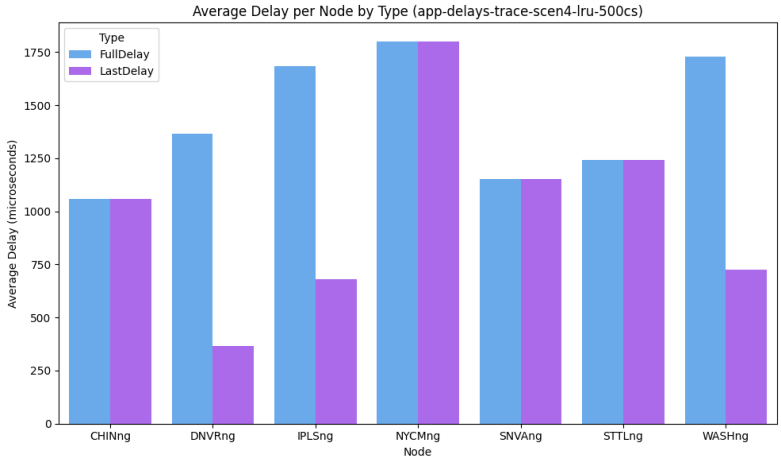

Figure A.10: Average delay per node for Scenario 4 LRU with 500 CS.

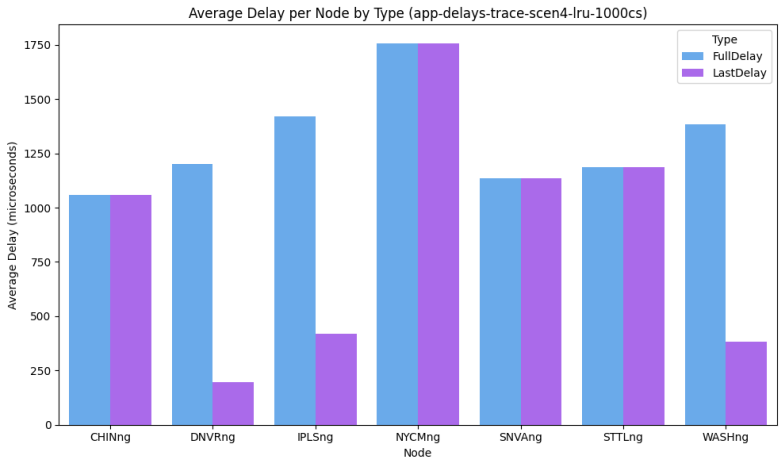

Figure A.11: Average delay per node for Scenario 4 LRU with 1000 CS.

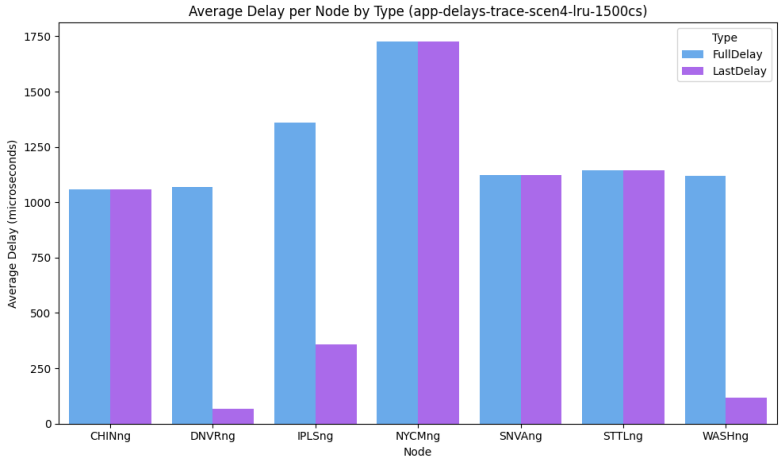

Figure A.12: Average delay per node for Scenario 4 LRU with 1500 CS.

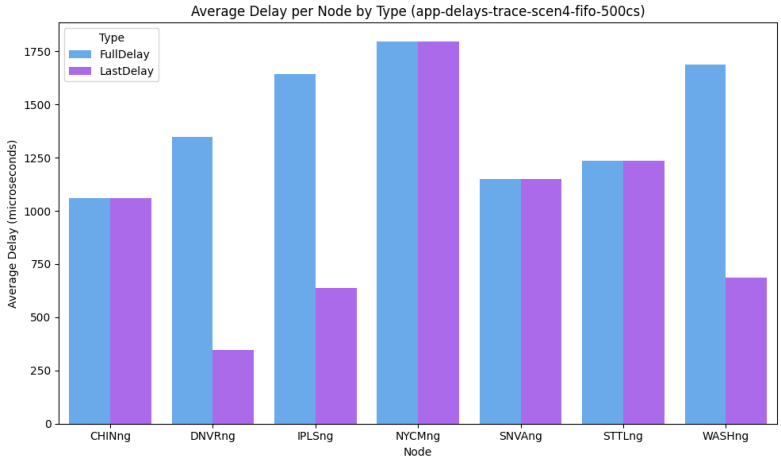

Figure A.13: Average delay per node for Scenario 4 FIFO with 500 CS.

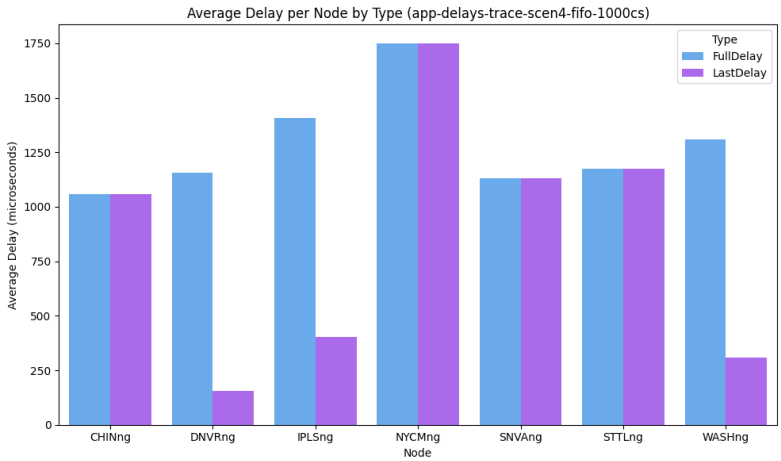

Figure A.14: Average delay per node for Scenario 4 FIFO with 1000 CS.

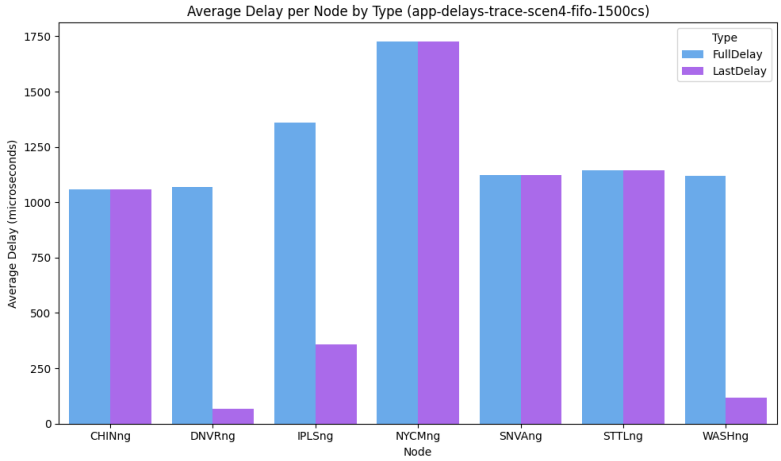

Figure A.15: Average delay per node for Scenario 4 FIFO with 1500 CS.

A.3 Path Reduction

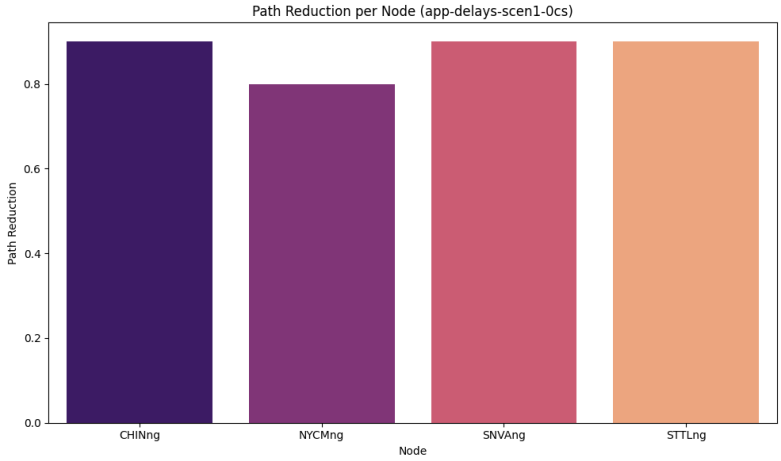

Figure A.16: Path reduction per node for Scenario 1.

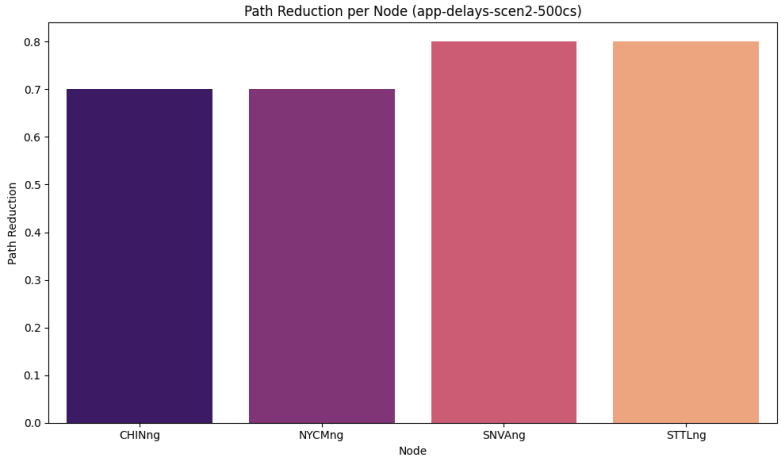

Figure A.17: Path reduction per node for Scenario 2 with 500 CS.

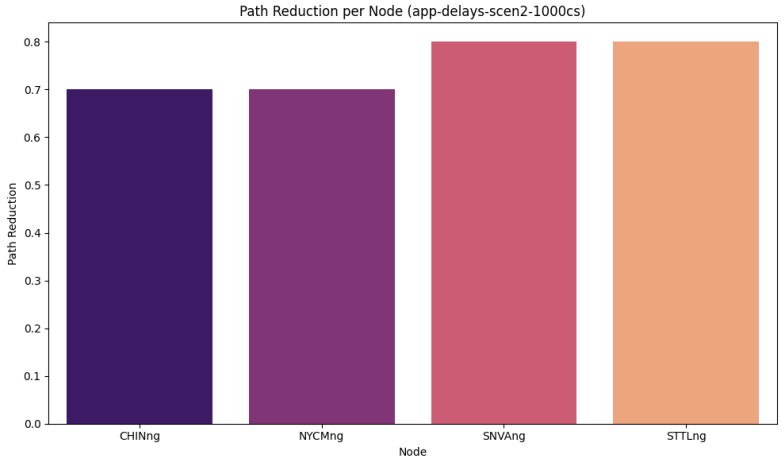

Figure A.18: Path reduction per node for Scenario 2 with 1000 CS.

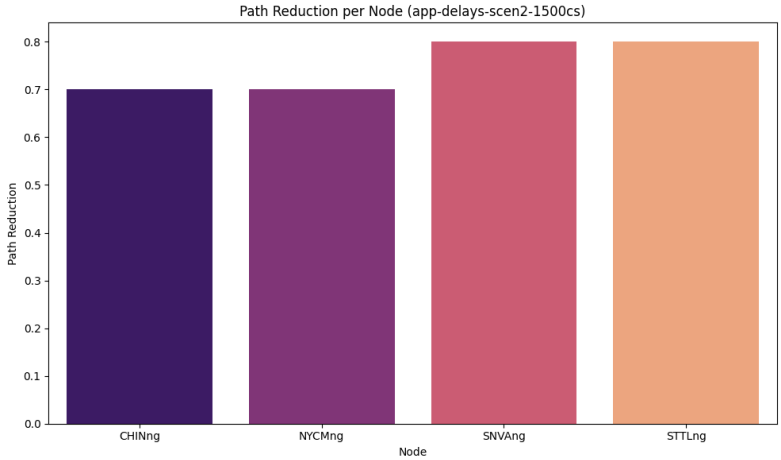

Figure A.19: Path reduction per node for Scenario 2 with 1500 CS.

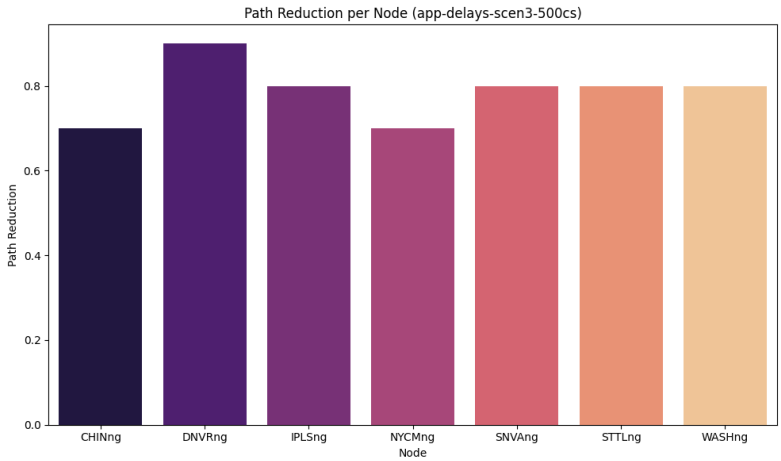

Figure A.20: Path reduction per node for Scenario 3 with 500 CS.

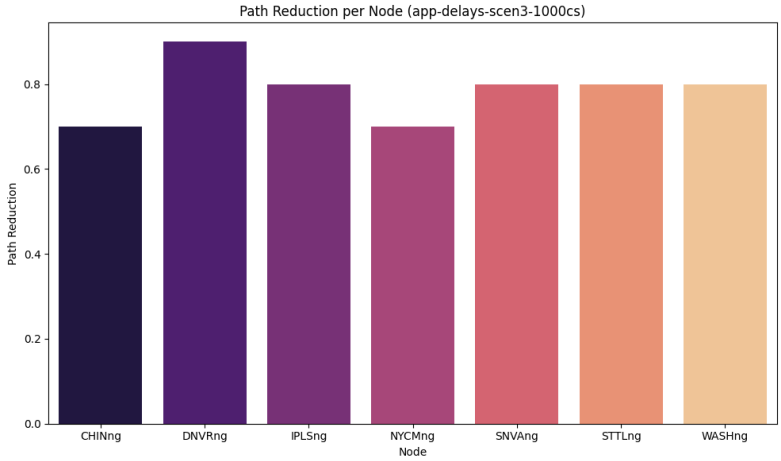

Figure A.21: Path reduction per node for Scenario 3 with 1000 CS.

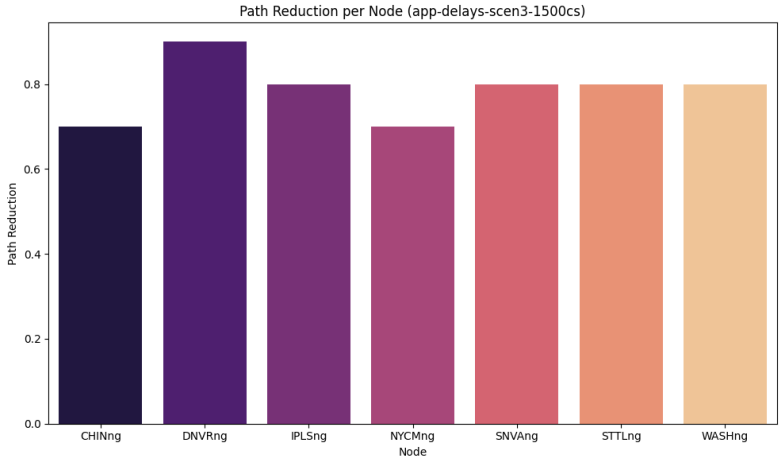

Figure A.22: Path reduction per node for Scenario 3 with 1500 CS.

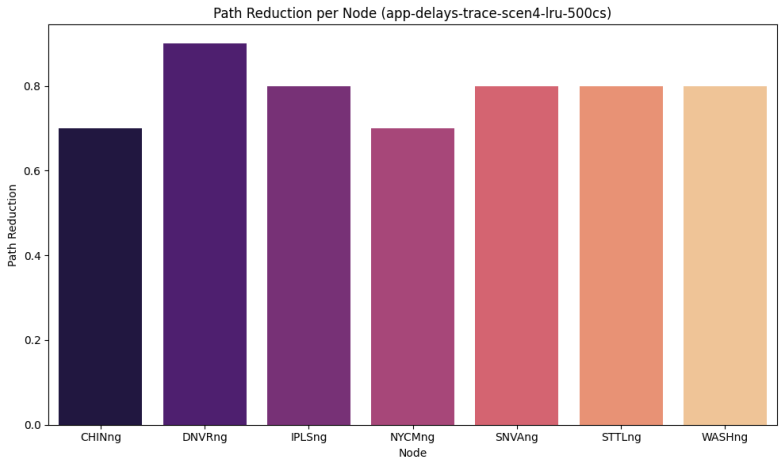

Figure A.23: Path reduction per node for Scenario 4 using LRU with 500 CS.

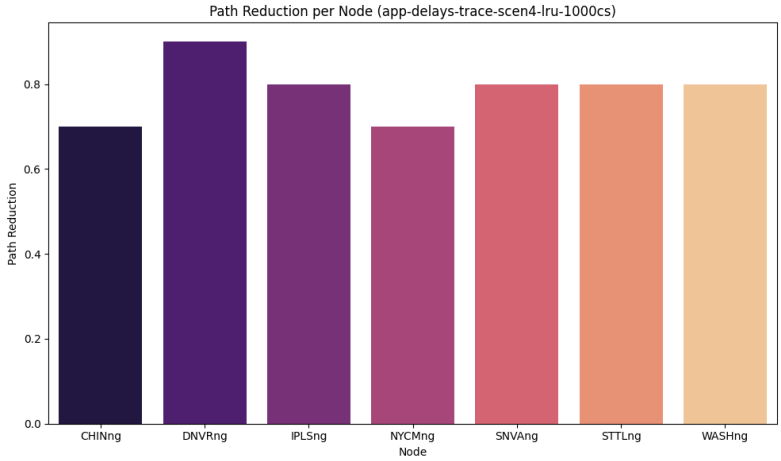

Figure A.24: Path reduction per node for Scenario 4 using LRU with 1000 CS.

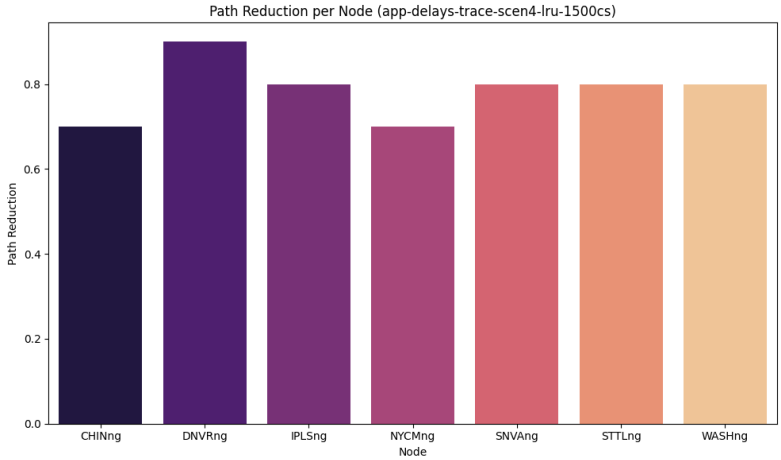

Figure A.25: Path reduction per node for Scenario 4 using LRU with 1500 CS.

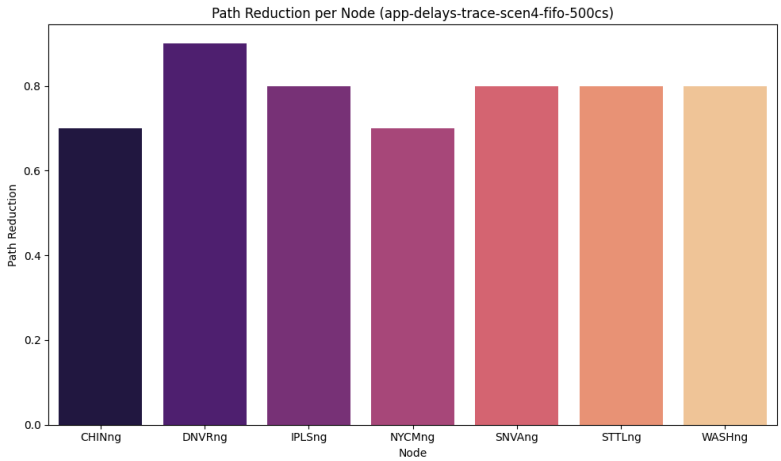

Figure A.26: Path reduction per node for Scenario 4 using fifo with 500 CS.

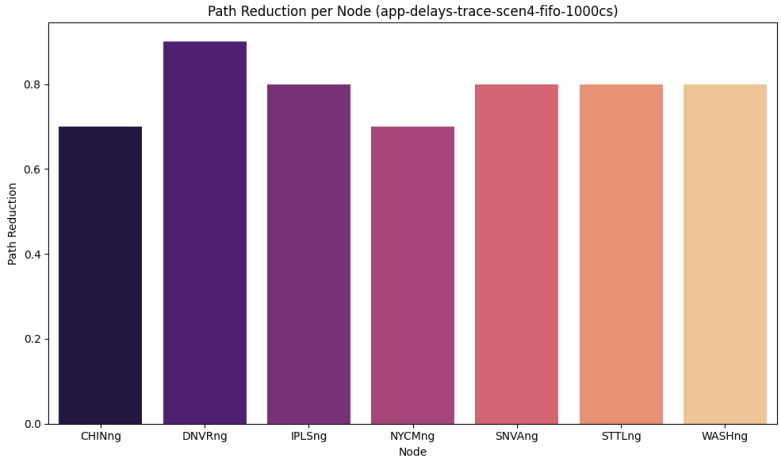

Figure A.27: Path reduction per node for Scenario 4 using fifo with 1000 CS.

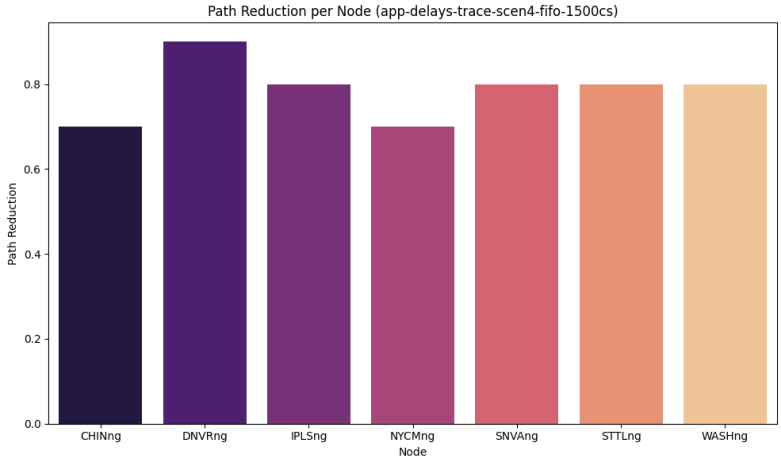

Figure A.28: Path reduction per node for Scenario 4 using fifo with 1500 CS.

A.4 Retransmissions

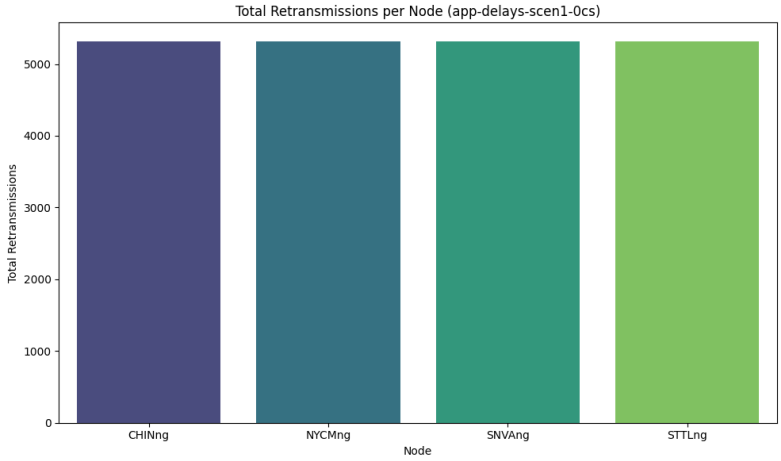

Figure A.29: Retransmissions for each node for Scenario 1.

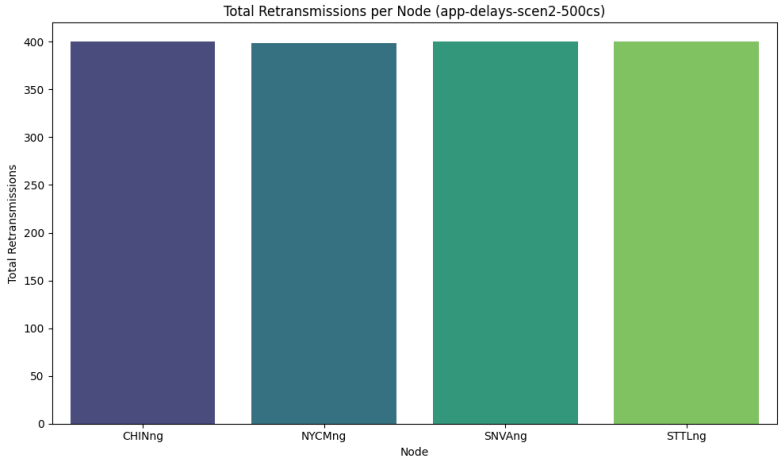

Figure A.30: Retransmissions for each node for Scenario 2 with 500 CS.

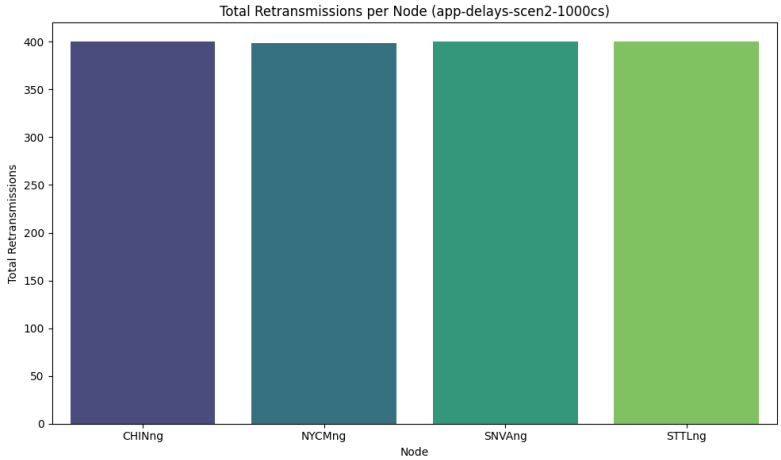

Figure A.31: Retransmissions for each node for Scenario 2 with 1000 CS.

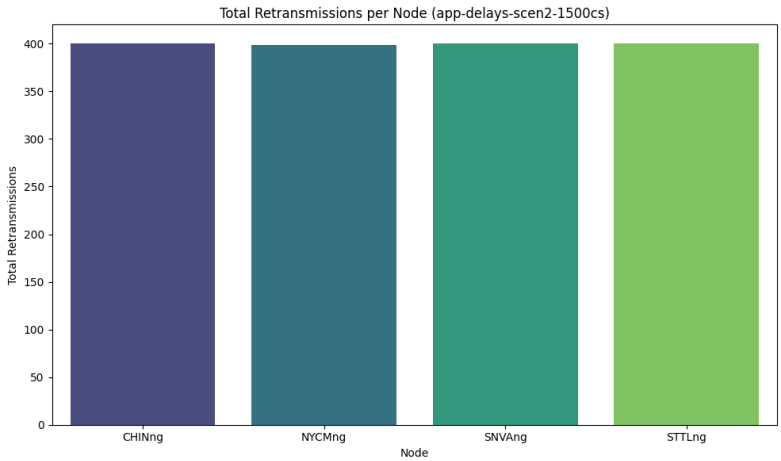

Figure A.32: Retransmissions for each node for Scenario 2 with 1500 CS.

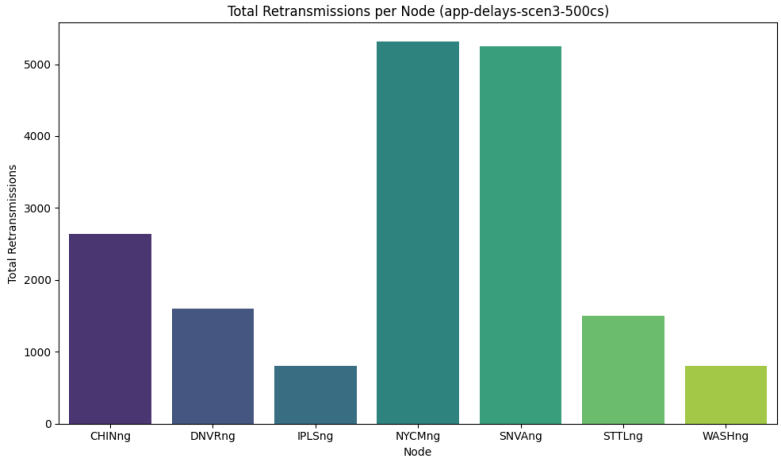

Figure A.33: Retransmissions for each node for Scenario 3 with 500 CS.

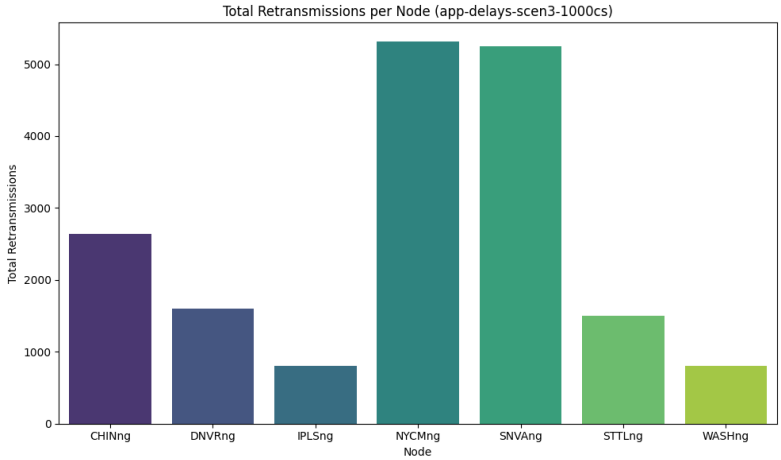

Figure A.34: Retransmissions for each node for Scenario 3 with 1000 CS.

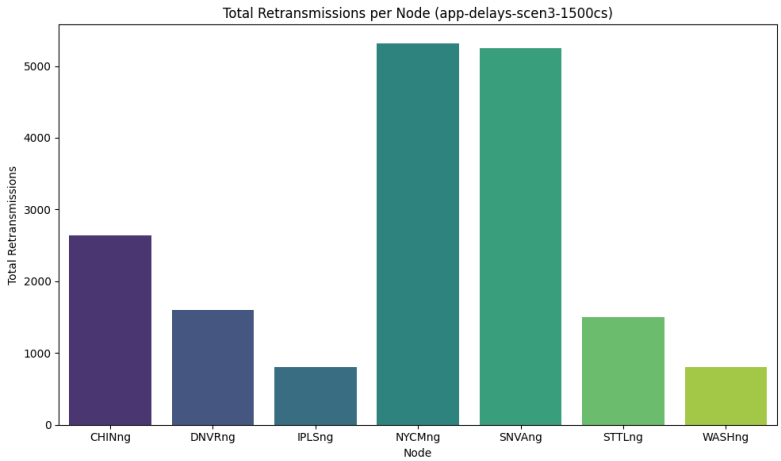

Figure A.35: Retransmissions for each node for Scenario 3 with 1500 CS.

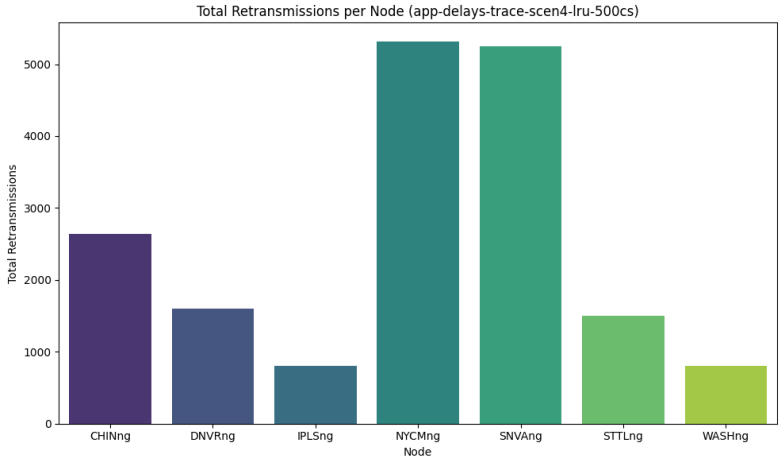

Figure A.36: Retransmissions for each node for Scenario 4 using lru with 500 CS.

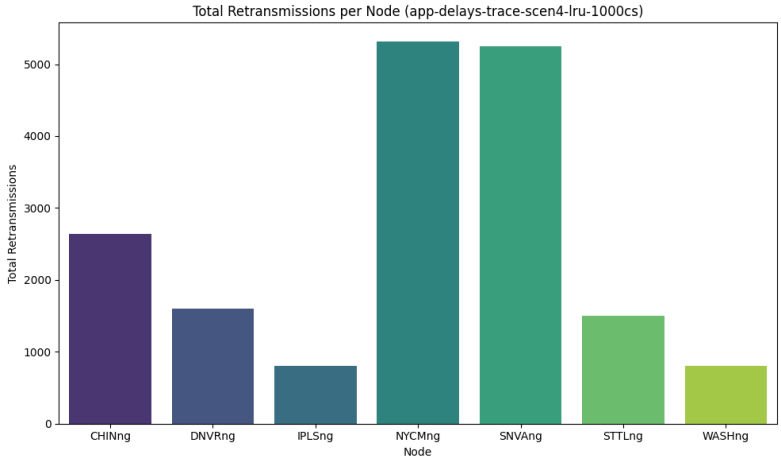

Figure A.37: Retransmissions for each node for Scenario 4 using lru with 1000 CS.

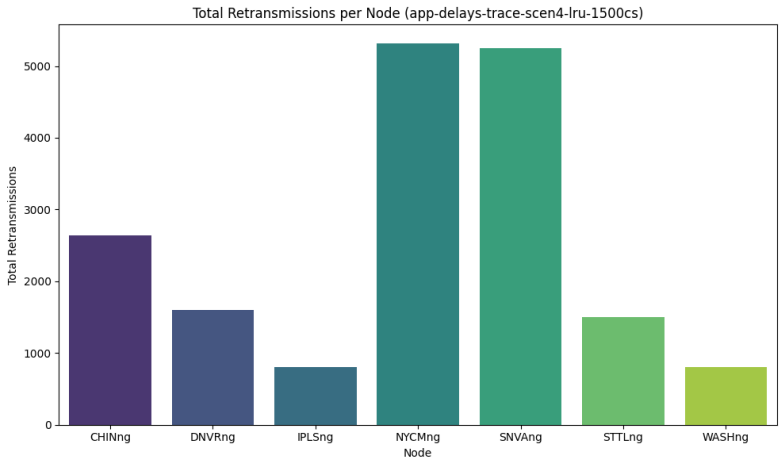

Figure A.38: Retransmissions for each node for Scenario 4 using lru with 1500 CS.

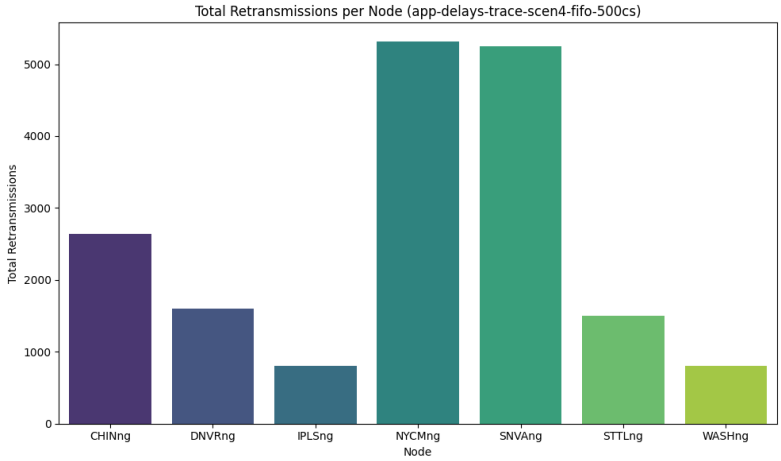

Figure A.39: Retransmissions for each node for Scenario 4 using fifo with 500 CS.

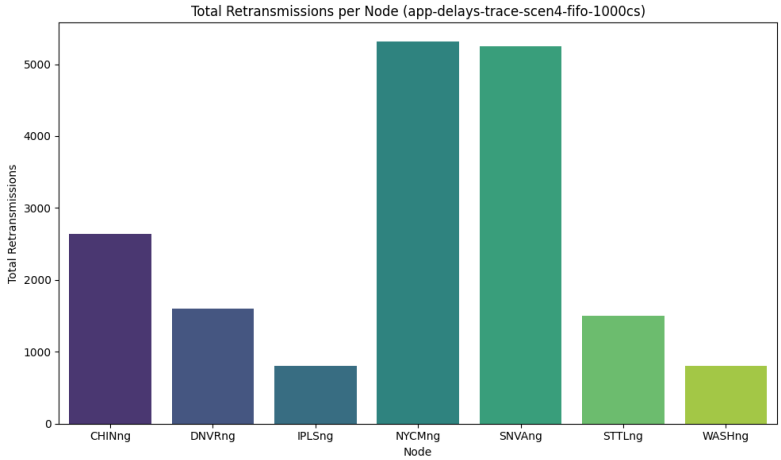

Figure A.40: Retransmissions for each node for Scenario 4 using fifo with 1000 CS.

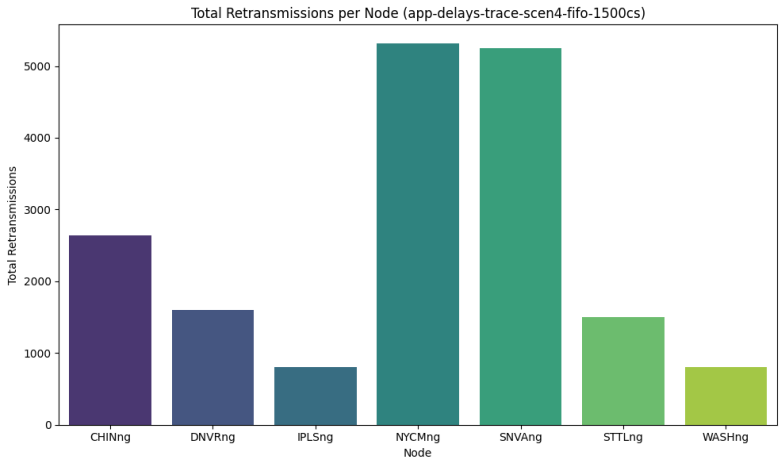

Figure A.41: Retransmissions for each node for Scenario 4 using fifo with 1500 CS.

A.5 Hop Count

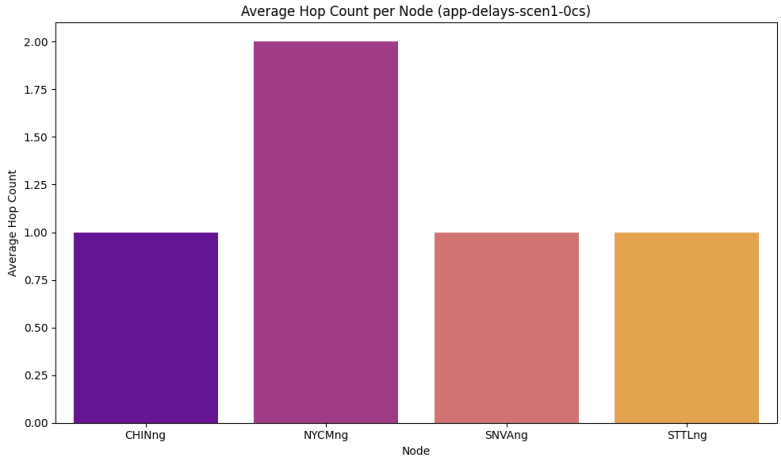

Figure A.42: Hop count for each node for Scenario 1.

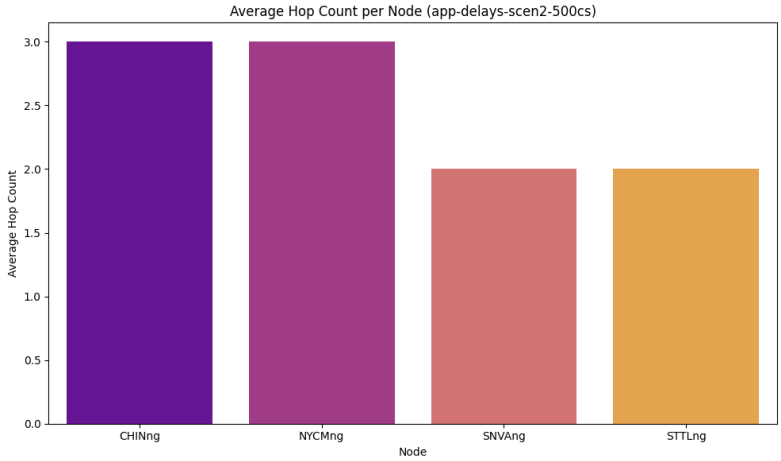

Figure A.43: Hop count for each node for Scenario 2 with 500 CS.

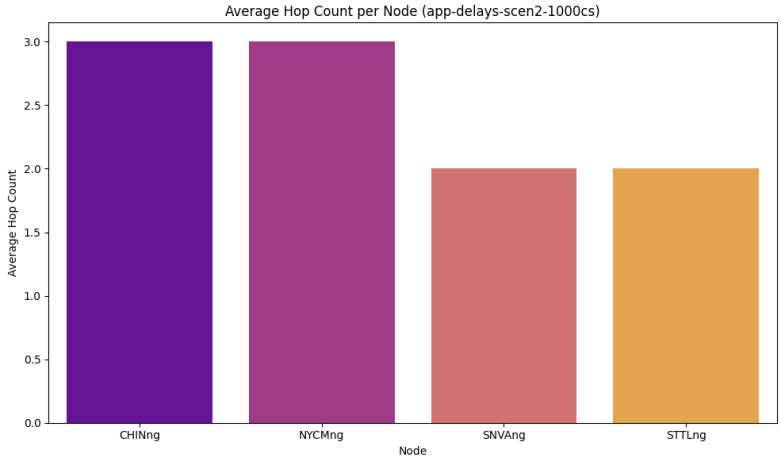

Figure A.44: Hop count for each node for Scenario 2 with 1000 CS.

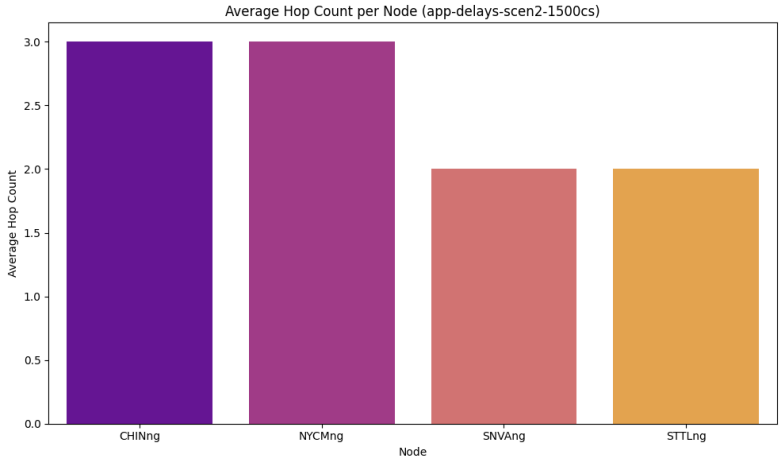

Figure A.45: Hop count for each node for Scenario 2 with 1500 CS.

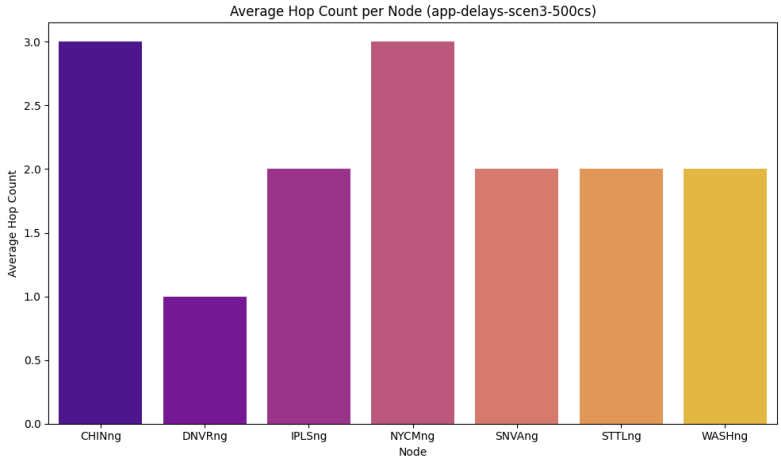

Figure A.46: Hop count for each node for Scenario 3 with 500 CS.

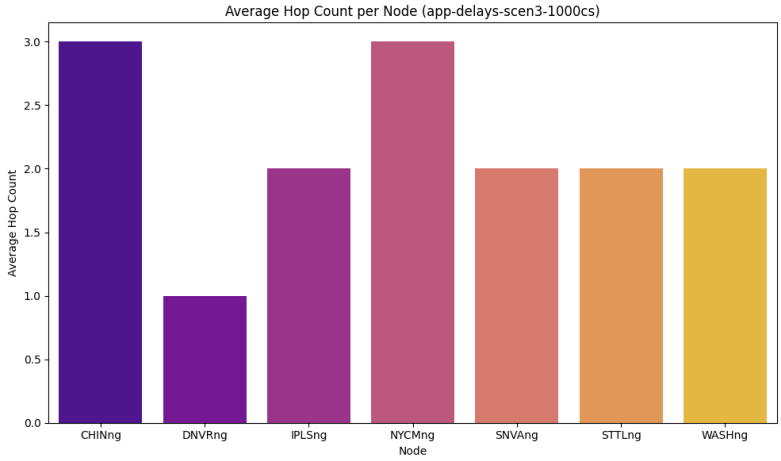

Figure A.47: Hop count for each node for Scenario 3 with 1000 CS.

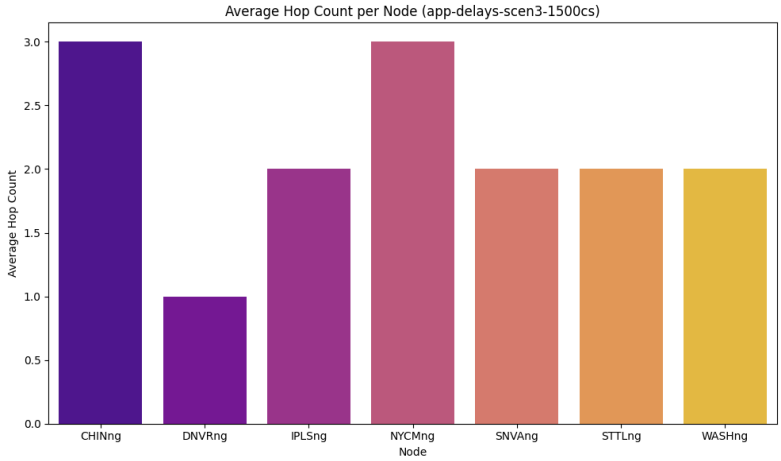

Figure A.48: Hop count for each node for Scenario 3 with 1500 CS.

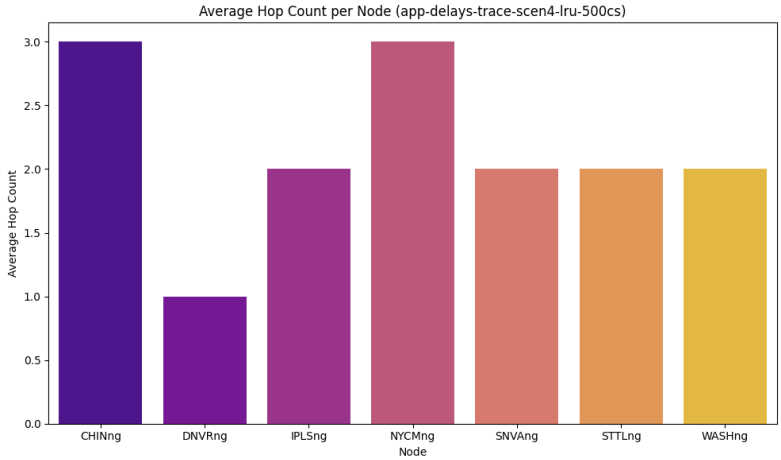

Figure A.49: Hop count for each node for Scenario 4 using lru with 500 CS.

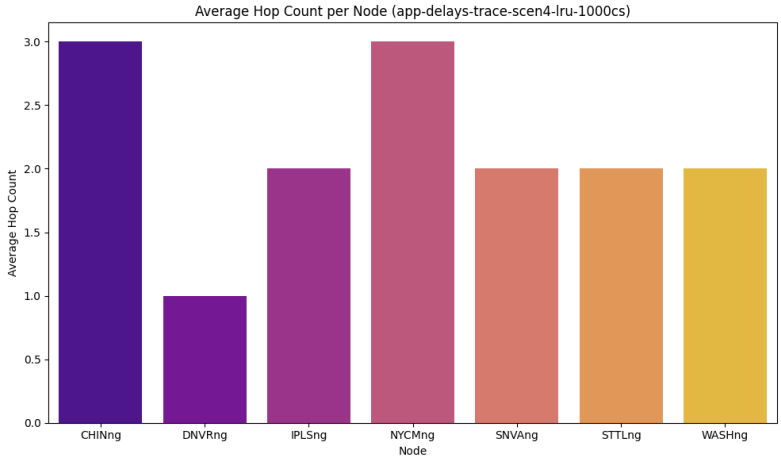

Figure A.50: Hop count for each node for Scenario 4 using lru with 1000 CS.

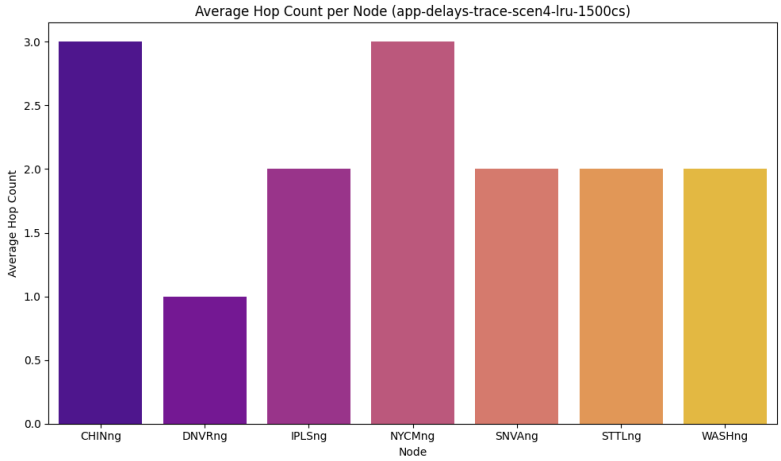

Figure A.51: Hop count for each node for Scenario 4 using lru with 1500 CS.

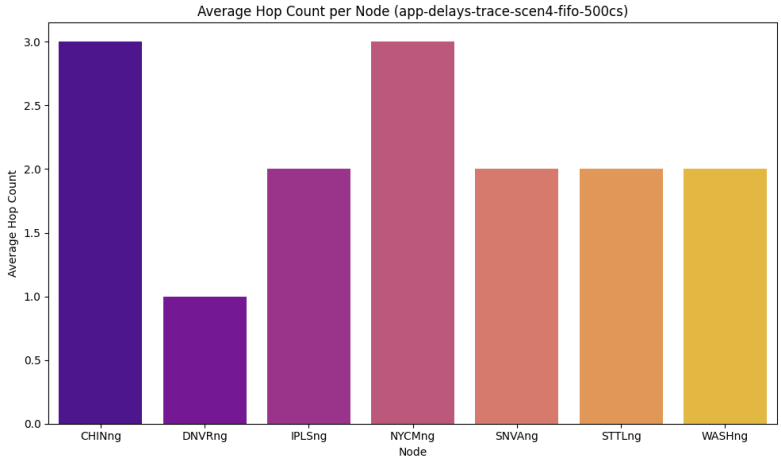

Figure A.52: Hop count for each node for Scenario 4 using fifo with 500 CS.

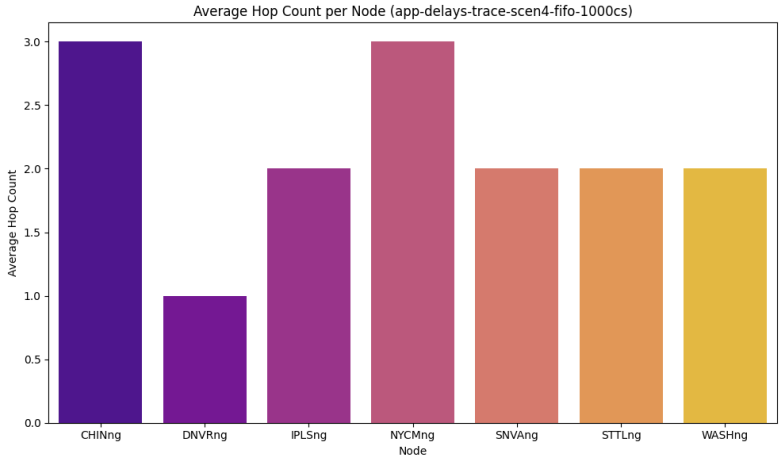

Figure A.53: Hop count for each node for Scenario 4 using fifo with 1000 CS.

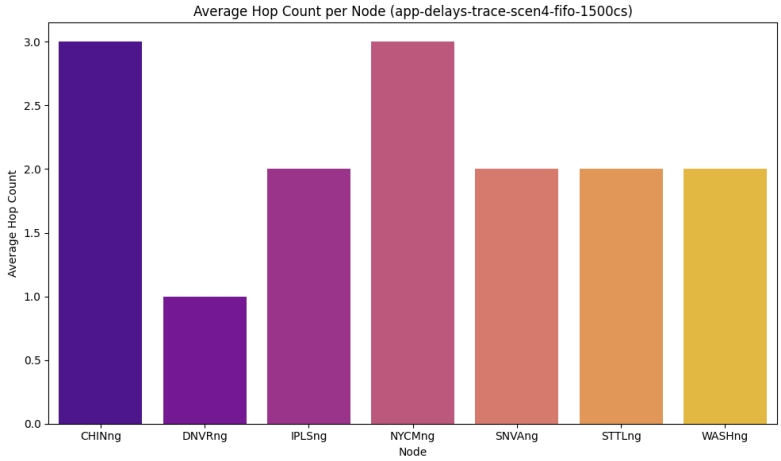

Figure A.54: Hop count for each node for Scenario 4 using fifo with 1500 CS.