Enhancing the Performance of Mobile Video Streaming Ecosystems

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Eman Ramadan Shehata

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Prof. Zhi-Li Zhang

December, 2022

Acknowledgements

There are so many wonderful people to whom I am really grateful for their support throughout my PhD journey. Each one played an important and special role.

Advisor. First and foremost I am very grateful to my advisor Professor Zhi-Li Zhang for his continuous support, invaluable insights, constructive feedback, and guidance during my PhD journey. I have learned a lot from him that made me the researcher I am today. His passion, devotion, and enthusiasm have always been a great inspiration to do great research work. He always had my back and supported me all the way.

Grants. Professor Zhi-Li Zhang has been relentless in submitting great proposals to make sure we have enough funds available to focus on research, and have every tool/device we need for our experiments. My research and travels to present my work and/or gain experience were supported by various fund resources: NSF grants CNS-1017092, CNS-1117536, CRI-1305237, CNS-1411636, CNS-1618339, CNS 1617729, CNS-1814322, CNS-1836772, CNS-1917424, CNS-1903880, CNS-1915122, CNS-1831140, CNS-1901103, CNS-2106771, and CCF-2123987. DTRA grants HDTRA1-09-1- 0050, HDTRA1-14-1-0040, DoD ARO MURI Award W911NF-12-1-0385, Raytheon/NSF subcontract 950001-2169/CNS-1346688, and a Huawei gift. Along with travel grants from ACM's Student Research Competition (SRC), GENI, N2Women, CRA-W, Arab Women in Computing (AWIC), and our department conference travel funds.

Committee. I am grateful to my PhD Thesis (or oral) committee members: Professors Jon Weissman, Andrew Odlyzko, Kangjie Lu, and Tian He for their feedback, fruitful discussions, and guidance on my research.

UMN Networking Lab. I was very fortunate to join such research group and build friendships with amazing students who were always willing to share their knowledge and exchange ideas for us to be better researchers. I'd like to thank Cheng Jin with whom

I explored the start of my PhD journey. I am especially grateful to Hesham Mekky and Arvind Narayanan for sharing with me useful programming tips and tricks that helped me during my PhD to do things efficiently. I'd also like to thank my collaborators: Braulio Dumba, Yang Zhang, Saurabh Verma, Pariya Babaie, Golshan Golnari, Taihui Li, (Ryan) Tu Nguyen, Xinyue Hu, Rostand A. K. Fezeu, Jason Carpenter, Udhaya Kumar Dayalan, Jacob Quant, Qingxu Liu, Rishabh Mehta, Wei Yu, Feng Tian, Peiqi Ji, Yu Liu, Tao Li, Ahmad Hassan, Nitin Varyani, Ziyan Wu, and Timothy J. Salo. We got to work on many exciting research projects together which enriched my knowledge and research experience.

CS&E Dept, CSE College, & UofM. I'd like to extend my gratitude all the CS&E faculty members from whom I learned a lot, especially Professors Feng Qian, David Du, Vipin Kumar, Ravi Janardan, and George Karypis. I definitely cannot praise enough our great administrators and staff, especially Georganne Tolaas, Luc Nelson, Joseph Nieszner, Mary Nissen, Phil Croteau, Leanne Siercks, Sue Gustafson, Aaron Connor, Irene Jacobson, Jeremy R. Jenkins, and Eric Kroetsch. They played a very important role, always did their best to make our life easier, and went above and beyond to help. I was honored to join several amazing groups (CSGSA, CS-IDEA, and CSE D&I Alliance), whose members are full of passion and have worked very hard to make our department and college a better place to study/work for everyone. My sincere thanks to Professors Mats Heimdahl, Daniel F. Keefe, Maria Gini, Loren Terveen, Mohamed F. Mokbel, Favonia, Victoria Interrante, Shana Watters, Edgar Arriaga, Mostafa Kaveh, Cara Santelli, David Odde, and Hannah Leopold. As well as great administrators, staff, and students especially Rhonda Zurn, Lindsey Fischer, Dan Garrison, Samantha Franco, Orla Gotthelf, Ellen Puffe, William Frazier, and Estelle C. Smith. I cannot thank the health-care team enough for taking good care of me throughout the years.

Alexandria University. I owe a debt of gratitude to the faculty members and my Master's degree advisors in the department of Computer and Systems Engineering in Alexandria University, Egypt. They did not only teach me the basics of Computer Science and academic research. Most importantly, they taught me the skill of learning new languages/tools on my own given my prior knowledge. I have been so lucky to meet lots of the department's brilliant alumni and students who enriched my knowledge and experience in the field.

Teachers. I can never forget all my school teachers in Egypt who taught me Math, Science, and English. Their efforts contributed a lot to where I am now. They helped me excel in these subjects, and definitely helped me a lot to build the basics for my communication and written skills.

Family. There are not enough words to describe my gratitude to my mom Samia and my dad Ramadan who supported my decision to travel abroad to pursue a chance for a better education, and for their selfless continuous support. My sister Amira who is the strongest person I know, someone I can always rely on. My brother Mohamed, his wonderful wife Shamiaa, my nephews Amir and Karim who I miss so much. My uncles, aunts, and their families who were always proud of my achievements especially uncle Amer. My late grandmother Fawqia who loved me since the day I was born, always believed in me to achieve all my dreams, and was so proud of what I have accomplished. It has been 4 years since she passed away, and I am sure she is in a better place watching me achieve this milestone.

UMN Family. This PhD journey would have been a lot harder without my second Egyptian family who made it feel like home, Reem Ali, Amr Magdy, Hebatallah Eldakiky, Ibrahim Sabek, Sara Aktham, Ahmed Samir, Sara Morsy, Hesham Mekky, Nora Ahmed, Ahmed Saber, May Elmofty, Omar Mehanna, their parents, and amazing kids who filled our lives with joy & pure love. We celebrated, struggled, and experienced many wonderful moments together forming a stronger bond between us, and great memories to cherish forever. They really filled a void for not having my family around during this time. I am also grateful to know many great Egyptian families and communities in CTC especially Dina Moussa, Hala Khalifa, and Maha Salah Zeedan. I was fortunate to have been surrounded by many great friends within the U especially Anisha Veeren, Ancy Tom, Ezgi Bozkurt, and many more who I played sports with.

Life-time Friends. I am very grateful to my friends across the oceans who have always believed in me to accomplish my dreams, and for their continuous love and support, Sally Nafea, Sarah Ibrahim, Mai Mohamed, Ragia Mamdouh, Golsa Daghighi, Nouran Zidan, Omnya Magdy, and all my amazing college and school friends.

Ellen. Last but not least, I'd like to thank Ellen DeGeneres for inspiring me to be healthier and more fit during this journey. For always having a positive attitude and never giving up no matter what. For her humor which got me through tough times and always managed to lift me up. I learned a lot from her perseverance, compassion, kindness, and giving back to others.

Minnesota. I am very grateful to the great state of Minnesota, which has been my home for all these years, with its nice people, lakes, and many indoor, outdoor and winter sports which I learned along the way and enjoyed very much. Even for its tough weather which made us stronger and able to do anything and go anywhere with proper preparation. Thank you Minnesota!

Thank you everyone, you definitely made an impact on my life, and I am forever grateful for your support.

Dedication

To my late grandmother Fawqia, my family members, and close friends who held me up over the years.

Abstract

Recent years have witnessed a rapid increase in video streaming services (*e.g.*, Netflix, YouTube, Amazon Video, ... etc) to meet users' interests as a result of the massive content published by content providers, high-speed Internet, the wide use of social networks, along with the growth in smart mobile devices. Additionally, the recent deployment of commercial 5G in 2019 and its potential for ultra-high bandwidth has enabled a new era for bandwidth-intensive networked applications such as volumetric video streaming. This growth in available content and demand places a significant burden on the Internet infrastructure. In addition to the complex structure of videos as each video is encoded in multiple resolutions, and different bitrate quality levels to support diverse end-user devices and network conditions. Thus, large-scale content providers have resorted to employing one or more content distribution networks (CDNs) to cache video content and handle user requests, as well as resorting to edge computing and machine learning to improve the performance perceived by their end users. Poor performance impacts user engagement, which leads to significant revenue loss for content providers. In this thesis, we discuss crucial research problems to improve the performance of mobile video streaming ecosystems to meet the scalability and user QoE performance requirements.

First, we study the performance of intermediate caches in a hierarchical cache network. We show that when cache servers at different layers act independently this leads to caching objects which are evicted before their next request arrives leading to cache under-utilization. To overcome this issue, we proposed "BIG" cache abstraction which deals with distributed cache pieces as if they are "glued" together to form one "virtual" "BIG" cache. Thus, allowing any existing caching strategy to be applied as a single consistent policy for this "BIG" cache. Consequently, "BIG" cache improves object hit probability, thereby minimizing the origin server load, and network bandwidth.

Second, object access patterns are frequently changing due to the frequent changes in object popularity due to its diurnal access pattern, and during its life span. Due to these frequent changes, caching algorithms cannot rely on the locally observed object access patterns for making caching decisions. On the other hand, manually tuning the caching algorithm for each cache server according to the changes in the request access patterns is very expensive and is not scalable. To address this issue, we developed a machine-learning LSTM Encoder-Decoder model for content popularity prediction. Our DeepCache is a self-adaptive caching framework for making end-to-end caching decisions based on the predicted popularity. We show that it manages to increase the number of cache hits for existing caching policies.

Third, routing is a central problem to ensure the resiliency of CDNs. Purely distributed routing algorithms such as Bellman-Ford suffer from the "count-to-infinity" problem, whereas Dijkstra's algorithm requires global topology dissemination and route recomputation. Much of the recent literature on resilient routing is resilient to k link/node failures for a constant k (and often placing topological constraints on the graphs), and none of them work under arbitrary link failures. To address this issue, we developed a proactive routing algorithm that ensures the connectivity between any pair of nodes under arbitrary failures without the need for global topology dissemination and route recomputation as in purely distributed routing algorithms. Our algorithm limits the number of nodes involved in the recovery process as well as the number of link reversals, and convergence time. An additional advantage is the ability to utilize multiple paths to send traffic between nodes due to utilizing directed edges between nodes even upon failures.

Finally, with the recent deployment of commercial 5G in 2019 and its potential for ultra-high bandwidth, we studied the characteristics of 5G throughput and its impact on video streaming applications. Our findings show that the wild fluctuations in 5G throughput and its dead zones lead to a large stall time while streaming videos. We redesigned video streaming applications to be 5G-Aware taking full advantage of the ultra-high bandwidth and overcoming its varying throughput. Our experiments show that our proposed strategies consistently deliver high video quality close to the theoretical optimal results reducing (if not eliminating) the stall time.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Content or information delivery is a major function of today's Internet. This content could be either static (*e.g.*, video, images, ... etc), or dynamic (*e.g.*, search responses, customized ads, ... etc) generated on the fly. In web content delivery, most of today's websites contain many objects (*e.g.*, images, scripts), which might be referenced by the main page, and by multiple other pages within these websites. When a certain page is requested, it is usually followed by requests for its related objects *(temporal locality of reference).* Thus, caching these objects, at web browsers and web caches at ISPs, helps minimize the latency experienced by users, and the consumed bandwidth.

Recent years have witnessed a rapid increase in video streaming services (*e.g.*, Netflix, YouTube, Amazon Video, ... etc) to meet users' interest as a result of the massive content published by content providers, high-speed Internet, the wide use of social networks, along with the growth in mobile devices. In 2020, mobile video streaming represented 65% of the global mobile downstream traffic [2], and is expected to reach 79% in 2022 [3]. This growth in available content and demand places a significant burden on the Internet infrastructure. In addition to the complex structure of videos as each video is encoded in multiple resolutions, and different bitrate quality levels to support diverse end-user devices and network conditions. Each video is divided into multiple chunks of few seconds (*e.g.*, 2 to 4 secs) for each resolution/bitrate level. Since no single server or even a data center has all the required storage, processing capacity and network bandwidth to process and serve all user demands, large-scale content providers have resorted to employing one or more content distribution networks (CDNs) to handle scalability, and improve the quality of experience (QoE) for their end-users.

CDNs contain a set of proxy servers distributed geographically, which can be either homegrown as in case of Google/YouTube [4], or commercial CDNs such as: Akamai, Level-3, and Limelight, used by Netflix [5] and Hulu [6]. CDNs are typically organized in a hierarchical structure, where the closest tier to users represents edge servers, and the farthest (at the top of the hierarchy) represents the origin server, which has a copy of all content objects, as revealed in [4, 5, 6, 7]. Requests from users are directed to the closest edge server. If the requested content is not cached, some complex techniques, such as DNS anycasting & redirections, HTTP redirection, and IP anycasting, are used to forward requests to higher layers in the hierarchy, till they reach the origin server if the requested objects are not cached anywhere else.

By 2022, 72% of the Internet traffic is expected to cross CDNs, up from 56% in 2017 [8]. Hence, adding cache storage space at routers becomes of utmost importance to handle this massive growth, and improve the network performance as well as user's QoE. Consequently, information centric networks (ICNs) (*e.g.*, NDN [9], DONA [10], CONIA [11]) have been developed as an emerging architecture for content delivery. ICN offers new primitives such as in-network caching, in which storage becomes an integral part of the network substrate (*i.e.*, routers have the capability to cache objects on-the-fly, and serve user requests for cached objects). Inefficient utilization of cache network storage and poor caching algorithms result in serving user requests from the backend/origin server, which increases the network load, as well as user-perceived latency. As a result, user engagement is impacted, which leads to significant revenue loss for content providers. *Thus, it is crucial to fully utilize the storage and processing capacities of all cache servers in a cache network efficiently to meet the scalability and performance requirements.*

Due to the urgent need for a much higher speed and lower latency for mobile video delivery, the 5th generation (5G) wireless technology made its debut as commercial services to costumers in early summer 2019. Most 5G deployments employ mmWave technology which promises to achieve a throughput theoretically up to 20 Gbps which is 100$\times$ better than today's 4G [12]. Hence, 5G has a potential to support several emerging bandwidth-hungry multi-media applications such as ultra-HD (UHD) 4K/8K, 360°, and volumetric (AR/VR) video streaming. Through several experiments with different settings, we found that mmWave 5G can in-deed offer ultra-high bandwidth (up to 2 Gbps) compared to less than 300 Mbps for 4G.

However, there are many practical issues with 5G, even with clear line of sight to the tower, 5G throughput exhibits much higher variation than 4G, mainly due to the PHY-layer nature of mmWave signal which can be easily blocked. This is even more severe when the user is moving, because of the frequent handoffs incurred due to obstructions caused by nearby buildings, vehicles, and other factors. Thus, the UE loses connection to 5G, switches to 4G, and the throughput even drops to zero (5G "dead zones"). For example, we experienced a total of 31 handoffs while walking a short 700 meters loop. This high amount of switching may confuse applications as it it is very hard to cope with the fluctuating throughput which may lead to highly inconsistent user experiences. This raises some important questions: 1) are current applications ready to take advantage of such high speed?, 2) how does 5G throughput variation actually impact the application performance?, and 3) whether applications need to be redesigned to cope with the 5G high speed and at the same time deal with its practical issues and fluctuations?

1.1 Outline and Contributions

The outline and the primary contributions of this thesis proposal are as follows and highlighted in Fig. 1.1 along with their challenges:

Cache Network Management Using BIG Cache Abstraction (Chapter 3)

From our analysis for the performance of intermediate caches in a hierarchical cache network, we found that they suffer from *thrashing* problem in which some objects are cached in these layers, and then directly evicted, before receiving another request, replacing more popular objects. The reason is caching servers at different layers act independently while taking decisions of which objects to cache and evict. Based on these observations, in this chapter we propose "BIG" cache abstraction which deals with distributed cache pieces as if they are "glued" together to form one "virtual" "BIG" cache. Thus, allowing any existing caching strategy to be applied as a single consistent policy for this "BIG" cache. Our findings show that "BIG" cache is able to *fully utilize all* caching resources at all layers regardless of the caching policy used, improves the object hit probability, and thereby minimizing the origin server load and network bandwidth.

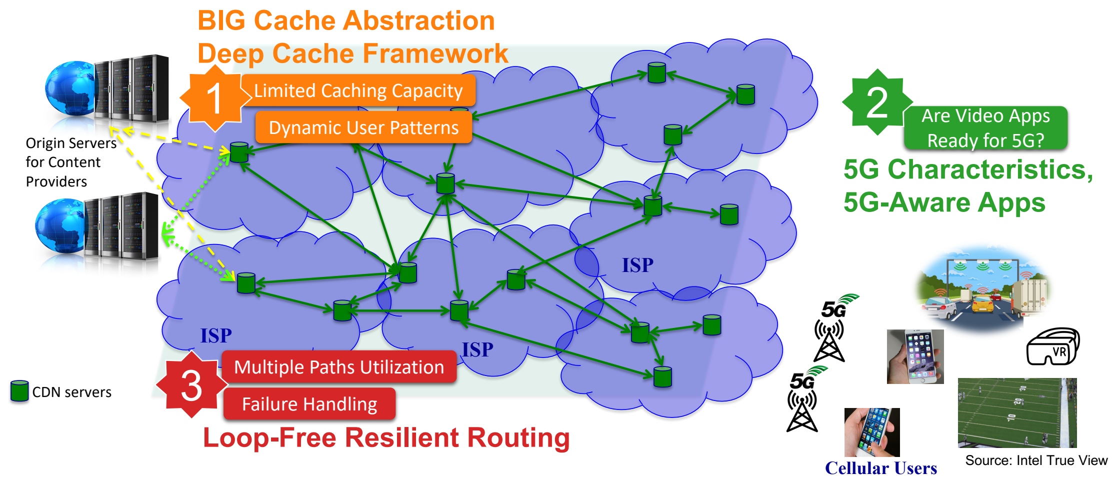


Figure 1.1: Future Mobile Video Streaming Ecosystem

This allows the cache network to scale by adding more cache resources and providing more content to be able to cope with the growing demand for content nowadays. Moreover, "BIG" cache eliminates the thrashing problem as caching decisions are not taken independently by each layer.

A Deep Learning Based Framework for Content Caching (Chapter 4)

Legacy caching policies decide which objects to cache purely based on the recent locally observed object access patterns which may lead to caching non-popular objects due to the lack of knowledge about future object popularity. Object access patterns are frequently changing because of the changes in object popularity due to its diurnal access pattern and life span. In addition, changes in request routing algorithms due to network/server failures can also cause changes in object access patterns. Due to these frequent changes, caching algorithms cannot rely on the locally observed object access patterns for making decisions. On the other hand, manually tuning the caching algorithm for each cache server according to the changes of request access patterns is very expensive and is not scalable. In this chapter, we recognize the problem of content popularity prediction as a seq2seq modeling problem, and to our knowledge, we are the first to propose LSTM Encoder-Decoder model for content popularity prediction. We develop our DeepCache framework for making end-to-end caching decisions based on the predicted popularity, and show that it manages to increase the number of cache hits for existing caching policies.

Resilient Routing (Chapter 5)

Routing is a central problem to ensure the resiliency of CDNs. With the increasing scale of networks, link or node failures are inevitable. Resilient routing, namely, the ability to continue routing operations without forwarding loops under failures is critical. Purely distributed routing algorithms such as Bellman-Ford suffer from the "count-to-infinity" problem, whereas Dijkstra's algorithm requires global topology dissemination and route recomputation. Much of the recent literature on resilient routing have been devoted to design *proactive* routing algorithms with *pre-computed* routing state (and limited *local* route exchanges or updates) that are resilient to k link/node failures for a constant k (and often placing topological constraints on the graphs). None of them work under arbitrary link failures. In this chapter, we develop a proactive routing algorithm that ensures the connectivity between any pair of nodes under arbitrary failures without the need for global topology dissemination and route recomputation as in purely distributed routing algorithms. Our algorithm limits the number of nodes involved in the recovery process, as well as the number of link reversals, and convergence time. An additional advantage is the ability to utilize multiple paths to send traffic between nodes due to utilizing directed edges between nodes even upon failures.

Mobile Video Streaming Using 5G Cellular Network (Chapter 6)

In this chapter, we focus on understanding the characteristics of 5G throughput and its impact on video streaming applications. Our findings show that the wild fluctuations in 5G throughput and its dead zones lead to a large stall time while streaming videos. Hence, we redesign the video streaming applications to make them "5G-Aware" to: i) take full advantage of the ultra-high bandwidth when it is available via *content bursting*, and ii) overcome wild bandwidth fluctuation and 5G "dead zone" by dynamically switching to 4G to maintain basic data connectivity. Our experiments show that these strategies consistently deliver high video quality close to the theoretical optimal results. Our study provides a baseline performance and identifies key research directions on how applications need to be developed to further improve their performance.

Chapter 2

Mobile Video Streaming Ecosystem Design Requirements and Challenges

There exist numerous challenges to design and develop a large scale mobile video streaming ecosystem. Starting from content generation with several resolutions and quality to support various devices and network bandwidth, to caching this huge amount of data, followed by strategies to deliver these videos which faces routing, scalability, and performance issues. Also, there are multiple key players involved such as content providers, cache network operators, cellular network infrastructure operators, and video streaming application developers. In this chapter, we highlight these challenges and the role of each key player.

Initially content providers need to decide whether they use their own CDNs, or employ one or more third-party commercial CDNs such as Akamai, Level-3, or Limelight. The content is then encoded into multiple resolutions with different bitrate quality levels in order to support various end-user devices, and also available user bandwidth depending on their connection link. The key goals of content providers are minimizing the load on their origin servers which host a copy of all content objects, and achieving a satisfied user's QoE to avoid revenue loss due to poor performance.

2.1 Caching Requirement and Challenges

As more videos become available, they need to be cached with higher quality and for different end-user devices. Caching objects closer to end-users minimizes their latency and improves the user's QoE. However, there is a limited caching capacity in any network.

Cache Utilization. Cache network operators (CNOs) care about full utilization of the available cache resources to cache as much content as possible and enhance user's QoE. If the requested content is not available in CDN cache servers, it needs to be retrieved from the original servers of content providers which increases user's latency. Thus, CNOs need to carefully decide which content to cache, which resolutions, and quality levels to encode for each original raw video. Having all possible variations of resolutions and quality levels for all videos is not possible due to the cache storage limits. On the other hand, if the requested resolution or quality level is not available, either a higher resolution/quality level is sent to the user which leads to stalls and more bandwidth consumption, or the user has to wait till the video is transcoded on the fly.

Changing Access Patterns. Another important challenge for CNOs is which caching policy to use which determines which objects to cache and evict at each server. If the content popularity is fixed, static caching is proved to be the optimal caching policy by staging the most popular content in edge servers close to users, and less popular content in higher caching layers closer to origin servers. However, content access patterns are always changing either within each day, or during its life span. Also, flash crowds are common when some videos become suddenly popular. Manual tuning for which objects to cache according to the monitored changes in request patterns is very expensive and not scalable. Thus, several reactive caching policies are used to monitor object's access patterns and then decide which objects to cache. However, these decisions do not consider future access patterns of each object, which can nowadays be learned and predicted through machine learning models as we propose in this thesis.

2.2 Bandwidth Requirement and Challenges

Extra high bandwidth is required to be able to support the emerging bandwidth-hungry multi-media applications such as ultra-HD (UHD) 4K/8K, 360°, and volumetric (AR/VR) video streaming. Even though 5G promises to deliver ultra-high bandwidth, but it also comes with its challenges.

Cellular Network Performance. Cellular network infrastructure operators need continuous monitoring for the performance of their service especially for 5G due to its wild fluctuation and dead zones. Hence, they need to collect real throughput measurements experienced by users to take decisions such as tower placements to enhance the received throughput and improve network coverage.

5G radios cover a broad range of frequency spectrum: low-band (<1 GHz), mid-band (1 - 6 GHz), and high-band (>24 GHz). The low-band frequency spectrum provides maximum coverage but limited bandwidth capacity, while high-band (mmWave) range can provide very high bandwidth capacity but its signals are highly sensitive and vulnerable to obstacles thus limiting its coverage. Between both these extremes lies the mid-band range, which provides a middle ground by providing higher bandwidth capacity than low-band with better coverage than that of high-band range. Commercial 5G deployments by carriers usually support a single class of frequency range. However, several carriers are now considering deploying multiple classes to leverage multiple frequency bands which is known as multi-band 5G. Hence, deploying multi-bands for 5G seems essential to ensure the advantage of extended coverage with an ultra-high bandwidth.

Designing Video Streaming Applications. Videos are usually divided into multiple chunks of several seconds, and each chunk is available in different quality levels. Adaptive video streaming means that the quality level of each chunk is specified based on the available bandwidth which is determined through the throughput prediction module in the video streaming application. Frequent changes of 5G bandwidth confuses applications due to its sudden change from 2 Gbps to 0, unlike 4G throughput. Thus, the previous assumption of relying on the current available bandwidth to specify the quality level leads to a very fluctuating user experience. Hence, video streaming applications need redesign to be "5G-aware" to utilize 5G's extra high bandwidth, and also cope with its wild fluctuations. In addition, once multi-bands are available, applications can utilize them to deliver videos; for example the mid-band(low-band) can be used to deliver the base quality layer, while high-band can be used to increase the quality level of buffered chunks before their play time if possible. Hence, guaranteeing at least the basic quality level is available at the buffer avoiding rebuffering events.

2.3 Latency Requirement and Challenges

Some 5G applications require minimum latency such as Autonomous vehicles (AV), AR/VR and Volumetric video streaming. Hence, the network needs to be robust and to recover quickly from any failure to support the requirements for such applications.

Resiliency. Directing user request's to the closest server which has a copy of the content has a great impact on the user's latency. Hence, CDNs usually deploy complex techniques such as DNS anycasting & redirections, HTTP redirection, and IP anycasting to forward user request to the closest server. However, server and link failures are very common and hence impact the user perceived latency based on how fast the failure is detected, and then the time to recover from this failure, and establishing a new path.

Scalability. User demands also change from time to time, and from one region to another. Hence, it is important for CNOs to monitor user demands and servers' load. For example, if a certain content becomes suddenly popular attracting more requests for a specific server increasing its load, it is crucial to be able to easily scale out the fingerprints of this object to include more servers to be able to serve the increasing demand without affecting user's QoE.

Chapter 3

Cache Network Management Using BIG Cache Abstraction

3.1 Introduction

Coping with this large-scale content delivery, *information-centric networks* (ICNs), especially NDN [13], has been developed as another emerging architecture for delivering content. ICN offers new primitives, such as pervasive caching, in which storage becomes an integral part of the network substrate (*i.e.*, routers have the capability to cache objects on-the-fly). ICN also advocates for routing requests, using *object names* (instead of IPs), to the nearest copy of content (nearest replica routing). When an object is found, it is cached *along-the-path* (*i.e.*, object replication) to the user, to be able to serve other requests for the same object (*a.k.a.* in-network caching). The effectiveness of these primitives have been questioned in some studies. For example, in [14], the authors concluded that the gain achieved over edge-caching and shortest path routing to the origin server, is minimal compared to the complexity associated with this new proposed architecture. However, with the continuous growth of the available content and user demands, edge servers alone can not be used to achieve a scalable and efficient content delivery system. Motivated by this, we have analyzed the performance of intermediate caches in a hierarchical network for content delivery, and realized that the poor performance is caused by the problem of *thrashing*, in which some objects are cached in these layers, and then directly evicted, before receiving another request, replacing more popular objects (see §3.2.2 for more details). The reason, behind this, is caching servers, at different layers, act independently, while taking decisions of which objects to cache and evict, without any coordination. Moreover, object requests received at intermediate layers are not independent of each other, and do not reflect the real object popularity, *e.g.*, requests for popular objects are served by lower layers, so they are not forwarded to higher layers. Thus, fully utilizing the available caching resources of a cache network is very important in designing large-scale content delivery system, because it affects the efficiency and scalability of the system.

In this chapter, we advocate "BIG" cache, as a new abstraction, for caching objects to fully utilize the available storage and processing capabilities of cache servers. Assuming requests for content are first received by edge servers. If the content is not cached, requests are forwarded, through a line of cache servers, along the path towards the origin server. In this proposed abstraction, intermediate cache servers, along the path from the edge server to the origin server, allocate (virtually) a portion of their caches to this edge server. These pieces are "*glued*" together to form one virtual "BIG" cache (see Fig. 3.4 and §3.4 for more details). Caching mechanisms are then applied to this virtual "BIG" cache, as if it is one single cache with a total capacity equals the sum of the cache sizes of these pieces. However, it is physically distributed among different layers. This means only one copy of any object is cached at any time along the path to the origin, and objects can move between cache boundaries according to the increase/decrease in their access rate. Evicted objects from one cache are added to the next layer, and are only evicted if they are removed from the highest layer. Thus, we can eliminate the thrashing problem (see §3.2.2 for more details). This leads to fully utilizing the storage of the intermediate caches, which increases the hit probability and thereby, minimizes user's latency, bandwidth, origin server load, and energy consumption due to unnecessary disk writes. Moreover, any existing caching policy such as: LRU, K-Hit, K-LRU, ... etc can be easily applied directly to this abstraction with an improved performance as indicated in §3.4.

In the following sections, we introduce the assumptions and the limitations of treating cache servers independently in §3.2 followed by related work in §3.3. In §3.4, we illustrate our notion of "BIG" cache abstraction, apply it to existing caching policies, and analyze their performance. In §3.5, we introduce an alternative caching policy

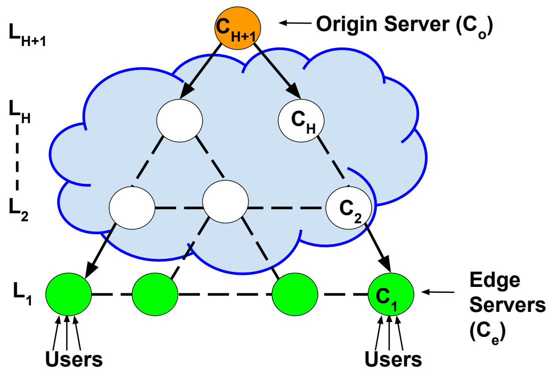


Figure 3.1: Network Model

dCLIMB for “BIG” cache, as a generalization of the CLIMB policy introduced in [15], to overcome the overhead introduced by some caching mechanisms after being applied to “BIG” cache. We address the problem of cache allotment for intermediate caches in §3.6. We present our evaluation in §3.7, discussion in §3.8, and the Chapter is summarized in §3.9.

3.2 Assumptions & Motivation

3.2.1 Network Model

We assume the cache network is modeled as a hierarchy of cache servers. Without loss of generality, consider the hierarchy is represented by a tree structure of $H+1$ layers. As shown in Fig. 3.1, the root at layer L_{H+1} represents the origin server C_o, leaf nodes at layer $L1$ represent edge servers C_e, and the intermediate servers are denoted by C_h, $2 \leq h \leq H$. The origin server has a permanent copy of the object collection consisting of N unique objects represented by $\mathcal{O} = \{O_1, O_2, \ldots, O_N\}$, of unit size each. The size of a cache server can be expressed in terms of the number of objects which can be cached[1]. Requests from a certain user populace, are directed to the closest edge server responsible for serving this populace. If the requested object O_i is cached, the object is returned; otherwise the request is forwarded along the path from the edge server to the origin server. When the object is found, it is returned back along the reverse path to the edge

[1]We use the symbol C_h also to refer to the cache size of layer L_h.

server. For each edge server C_e, requests follow the standard *independent reference model* (IRM) (*i.e.*, requests are independent of each other). The rate of requests for object O_i is denoted by λ_i, governed by a Poisson process, and the total request rates for all objects is $\lambda = \sum \lambda_i$. The access probability of each object is $a_i = \lambda_i/\lambda$, following a Zipf distribution with parameter $\alpha \in [0.6, 1]$, where a_i is proportional to $\frac{1}{i^\alpha}$, and $\sum a_i = 1$. Without loss of generality, we assume objects are ranked by their popularity; O_1 is the most popular object, and O_N is the least popular. Request rates follow object popularity (*i.e.*, $a_1 \geq a_2 \geq \cdots \geq a_N$). In general, same objects can be requested from different edge servers, but they are independent, and their request rates might not be the same (*i.e.*, a popular object in a user populace might not be popular among other user populaces).

Through this chapter, we differentiate between two terms: object allocation and caching policy. *Object allocation* is used to specify when a requested object is found, how it is cached on its way back to the user, *e.g.*, "cache everywhere" means the object is cached at all intermediate servers on its return path. While "edge caching" refers to caching the object at the edge server only, and intermediate servers just pass the object to the previous layer without caching it. *Caching policy* (such as LRU, K-Hit, ... etc) is used to specify which object to cache, how the object is cached in a single cache (*i.e.*, its order among other objects within the cache), and when the cache is full which object to evict. We compare different object allocation strategies, and caching policies according to the following metrics. Assuming a line of caches starting from an edge server C_e to the origin server C_o, and passing by a set of intermediate caches C_h, $2 \leq h \leq H$. The hit probability for object O_i at the h^{th} layer cache server is denoted by p_{ih}, which represents the percentage of requests of object O_i served by cache server C_h. The overall hit probability of O_i, being served from any layer other than the origin server, is calculated by $\sum_{h=1}^{H} p_{ih}$. The overall hit probability at a cache server C_h is calculated by $\sum_{i=1}^{N} p_{ih}$, which is equal to the cache size, if the caching policy is fully utilizing the available cache space. The percentage of requests served by the origin server is calculated by $1 - \sum_{i=1}^{N} \sum_{h=1}^{H} p_{ih}$. Higher origin server load affects the cache network scalability. Finally, given the latency of serving a request from cache server C_h as ϕ_h, the latency for each object O_i, and the overall latency of the system can be computed.

3.2.2 Problem of (Cascade) Thrashing

ICN suggested “in-network caching” helps maximize the probability of sharing, which in turn minimizes both the upstream bandwidth demand and the downstream content delivery latency. Hence, in this section, we study the problem of object placement in a hierarchy of distributed cache servers and its effect on cache utilization and performance. Using simulation, we found *thrashing* happens when objects are cached and directly evicted before receiving their next request. Upon frequent thrashing occurrences, the cache efficiency is minimized due to its under-utilization. Thrashing also has a cascaded effect, when unpopular objects are cached at lower layers, they force other (more popular) objects to be evicted, which may be cached at other layers, leading to more evictions. We illustrate the utilization of the intermediate layers of a tandem hierarchy of cache networks by applying the current caching policies, which have been well studied over the past decades, such as: LRU, K-Hit, ... etc

In this simulation, we use “leave-copy-everywhere” (LCE) as the object allocation strategy, which means when a request for object O_i is served by cache layer $C_{h'}$, other caches on the way back to the edge server $C_h, 1 \leq h < h'$ will cache a copy of this object. When the cache is *full*, LRU cache replacement policy is used to evict the least recently used object. We use $H = 4$, where C_1 is the edge server, and C_5 is the origin server offering $N = 100$ objects of unit size. The size of each cache $C_h = 10$, $1 \leq h \leq H$. The access probability of each object follows Zipf distribution with $\alpha = 1.0$, and the simulation lasts for $R = 1M$ requests.

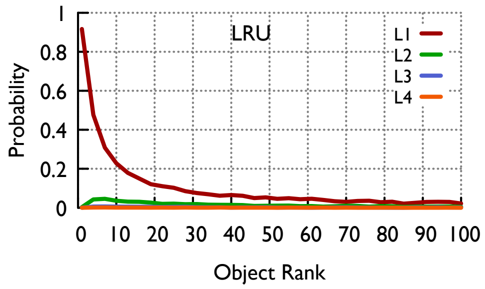


Figure 3.2: LCE-LRU - Object Hit Probability $N = 100$, $Cache\ size = 10$, $H = 4$, $R = 1M$

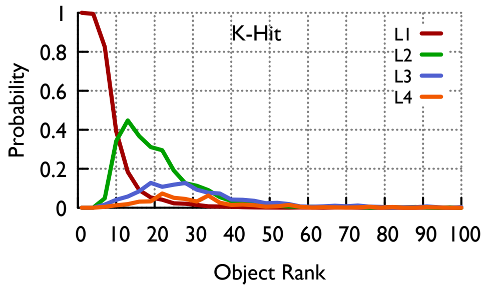


Figure 3.3: K-Hit - Object Hit Probability $N = 100$, $Cache\ size = 10$, $H = 4$, $R = 1M$, $K = 4$, $timer = 500$

Fig. 3.2 shows the object hit probability for the edge server $L1$ and the intermediate layers $L2$ through $L4$. We can notice that at $L1$, the 5^{th} most popular object is served only 40% from the edge server, indicating high eviction rate among the popular objects due to thrashing. The intermediate layers are poorly utilized, and are not able to serve more than 5% of the requests received for any object. That's why the authors in [14] have suggested that caching only at edge servers leads to the same performance, because the hit probabilities of objects at intermediate layers are very low. Using a more sophisticated caching policy as K-Hit, in which at each cache, objects are cached after receiving at least K requests within a predefined time period. When the cache is full, the object with the least number of hits is evicted. Fig. 3.3 shows the object hit probability for the edge server and intermediate layers when $K = 4$. We can notice the improvement in the hit probability, however it is still limited. Moreover, as h increases, the improvement diminishes drastically as we can see at $L3$ and $L4$. This can be further improved by increasing the value of K, however, this affects its adaptation to changes in traffic pattern.

Hence, we can notice when "caching-along-the-path" and LRU are used, intermediate cache layers in a tandem cache network perform poorly. This poor performance happens because each cache server in the hierarchy operates *independently* and decides on its own which object to cache and evict. This independence has two main issues; first the request arrival pattern seen by each cache server (*i.e.*, layer) does not reflect the real popularity of objects, as requests for popular objects served by lower layers are filtered. Thus, their access rate at higher layers can't be used to estimate their actual popularity. Second, an evicted object is simply discarded and its past access information is lost. The combination of these issues leads to the cascading effect of thrashing and under-utilization of the available cache resources.

More importantly, cache under-utilization leads to more requests being served from the origin server, which requires more capacity and bandwidth. This defeats the purpose of employing CDNs with hierarchical structure to minimize the origin server load and allow for system scalability. Moreover, content providers pay for these cache resources, however, they are being under-utilized. Finally, wasteful operations of caching objects that are directly evicted consumes disk energy, which is becoming an important aspect, while designing CDNs as mentioned in [16].

3.3 Related Work

Caching has been well studied in the past, especially for single cache. Due to the growing interest in content, especially video content, large-scale content providers rely on CDNs, because no single server, or even a data center, has the processing capability, or the network bandwidth required to serve all user requests. CDNs contain a set of proxy servers distributed geographically, which are typically organized in a hierarchical structure, as revealed in [5, 4, 7]. In addition, the interest in caching techniques has increased with the development of ICN networks, where caching becomes an ubiquitous part of each router. Thus, the performance of ICN depends on its caching mechanism. Hence, it is important to develop caching mechanisms for hierarchical structure. Moreover, it is essential to develop tools to be able to analyze the performance of these interdependent caches. Next, we discuss the limitations of the related work relevant to these two aspects.

ICN offers new primitives, such as: "in-network caching" or "caching-along-the-path". Our results in §3.2.2 shows that "in-network caching" leads to poor performance at the intermediate layers. However, we do not claim a novel contribution about the poor performance of various caching techniques at the intermediate layers, as it was suggested by anecdotal evidence previously in [17, 18]. That's why the authors in [14] advocated the idea of "caching at edge servers only". Caching content close to end users minimizes user's latency, and reduces the network bandwidth. However, the only drawback is that caching at edge servers only can't be used to achieve a scalable and efficient CDN, due to the massive growth of available content, especially video content with multiple chunks of different bitrates.

As an intermediate solution between "caching everywhere along the path" and "edge caching", the authors in [19] have presented some other object allocation strategies, such as: "leave copy probabilistically" (LCP), and "leave copy down" (LCD). In LCP, if an object is found in layer $C_{h'}$, it is cached on its way back to the user with a probability q for each cache C_h where $1 \leq h < h'$. In LCD, the object is only cached at the lower layer $C_{h'-1}$. However, these strategies still do not fully utilize the available storage at the intermediate servers, as there is at least one additional copy of objects is stored. This is indicated by the results presented in §3.4 for tandem network, and §3.7 for hierarchical tree network. In summary, applying caching policies and object allocation techniques independently, without any cooperation, leads to object replication at the cache network, consuming the available storage, without any performance gain.

In order to handle the problem of independent decisions, a centralized controller can be used to make decisions for all caches, relying on the collected statistics about the requests received at each cache. Due to the global view and the centralized decision making, this solution is expected to utilize the cache resources at each layer better. However, it might not be scalable as each cache needs to refer to this controller for each request to know whether it needs to cache the requested object or not. This centralized controller also represents a single point of failure; if it fails, caches won't be able to perform. Thus, we need caches to run a distributed algorithm, and at the same time coordinate together. Therefore, in this chapter, we advocate for the notion of "BIG" cache abstraction, and show that storing only one copy along the path is enough. It leads to fully utilizing the available storage, especially at the intermediate caches, and also helps improve the overall network performance for both the origin server's load and the user's latency, as we expound in the next section. We believe, to the best of our knowledge, that we are the first to introduce this cache abstraction.

On another aspect, the possibility to theoretically analyze and evaluate the performance of cache networks is of equal importance. This is a hard task, due to the interaction and interdependency between layers; during fetching objects from caches along the path to the origin server, or caching them on the way back to the user. For instance, assuming a hierarchical tree cache network, the requests arrival rate at a higher layer, depends on the miss rates of the lower layers. Therefore, the requests arrival rate at the first layer, can't be used for higher layers as it is. Over the years, several approximations [20, 21, 22, 23] have been proposed to predict the cache performance, as it is not easy to exactly estimate the requests arrival rate at higher layers. The common approach in these approximation techniques is described as following: the average request rate for object O_i at a specific cache C_h is calculated, if the hit probability of object O_i is known at all other caches forwarding their missed requests to C_h. The requests arrival process for each object O_i at any cache C_h is assumed to be Poisson. Thus, the hit probability of each object O_i at each cache C_h can be calculated independently under the IRM assumption. However, assuming the arrival process is Poisson at each cache is not accurate, as shown by [22], which discusses the error introduced using such technique.

Another alternative approach has been recently proposed in [24], for TTL-based eviction policies, which relies on Che's approximation [21] to calculate the characteristic time T_{C_h} for each cache C_h, which specifies the average eviction time of any object O_i at each cache C_h. In order to specify the request rate of object O_i at a higher layer C_h (*i.e.*, non-ingress cache), the miss process/stream of each cache forwarding its missed requests to C_h is characterized as a renewal process. Then, the inter-request time distribution of object O_i at C_h is calculated by the superposition of the independent renewal processes forwarding their traffic to C_h. However, this approach only analyzes LRU, RANDOM and FIFO caching policies. Also, when this approach is applied to larger networks, it requires intensive computation.

The authors in [19] have presented a different approximation. They analyzed various caching techniques, such as: LRU, q-LRU, K-LRU, ... etc, for single cache, under IRM and renewal process request models, using different object allocation strategies, such as: LCE, LCD, and LCP. Then, they extended their analysis for tandem networks by characterizing the request arrival process at higher layers as an ON-OFF modulated Poisson. The ON state happens, when the requests are directly received at the higher layer, as the object is not cached at the lower layers. The OFF state happens, when the requests are not received at the higher layer, because they are served by the lower layer where the object is cached. Then, they propose how this approach can be generalized to any network by defining the relation between the average arrival rate of requests for object O_i at cache C_h and the hit rate of object O_i at lower layer caches. Finally, the hit probability can be calculated using a conditional probability, given that object O_i is cached at C_h, when a request is forwarded from another cache to C_h, depending on the characteristic times of these two caches (see [19] for more details).

The proposed approach [19] analyzes various caching techniques, and can be generalized for arbitrary topologies. However, it is still only applicable under IRM process model. Thus, it can't handle bursty of requests and renewal processes, which are possible scenarios in any CDN. Analyzing such cases could be possible and easier for single cache, without the interdependency between different cache layers. Our proposed "BIG" cache abstraction has another advantage is that it makes the theoretical performance analysis more manageable. In “BIG” cache, existing caching policies can be directly applied to the virtual “BIG” cache as a single consistent policy. Hence, the analysis for single cache can be directly applied to this “BIG” cache as illustrated in §3.4.1, without the need for a complicated analysis to handle the interaction between cache layers.

3.4 “BIG” Cache Abstraction

Conventionally, in a hierarchical cache network, caching strategies are applied *independently* at each layer, without any interaction between caches at different layers. This leads to the problem of thrashing, which results in poor performance of cache network, especially at higher layers as shown in §3.2.2. Thus, in this section, we present the notion of “BIG” cache abstraction in which caches from different layers are considered as one virtual “BIG” cache, which is formally defined next. Then, we show, by theoretical analysis and simulation, when “BIG” cache abstraction is applied to existing caching policies, it enhances their performance rather than when these caching policies are being applied independently at each layer.

Given a hierarchical cache network, where edge caches $C_e = C_1$ lie at the first layer, and the origin server $C_o = C_{H+1}$ lies at the root of the hierarchical tree. The path from each edge server to the origin server passes by a sequence of intermediate caches $C_2, C_3, \ldots C_H$, which are shared by several edge caches. By partitioning each intermediate cache server according to the number of edge servers sharing it, such that C_h^e represents the share of each edge cache, where $\sum C_h^e = C_h$, $e \in \mathcal{E}$ (the set of edge servers), and $2 \leq h \leq H$, we can form a tandem cache from each edge server C_e to the origin server C_o, using its corresponding cache piece from the intermediate servers, as shown by the different colors in Fig. 3.4. For each edge server, we consider each cache piece from the edge server to the origin server $C_1^e, C_2^e, C_3^e, \ldots, C_H^e$ as if they are *glued* together to form one virtual “BIG” cache, with a total capacity $C^e = \sum C_h^e, 1 \leq h \leq H$. For simplicity, we will drop the superscript e, when we consider a tandem cache network.

Any existing caching strategy, such as LRU, K-LRU, q-LRU, FIFO, ... etc (see [19] for more details) can be applied consistently to this “BIG” cache as a single caching policy by allowing objects to move across boundaries, as opposed to H independent caching policies. Evicted objects are discarded only if they are removed from one layer

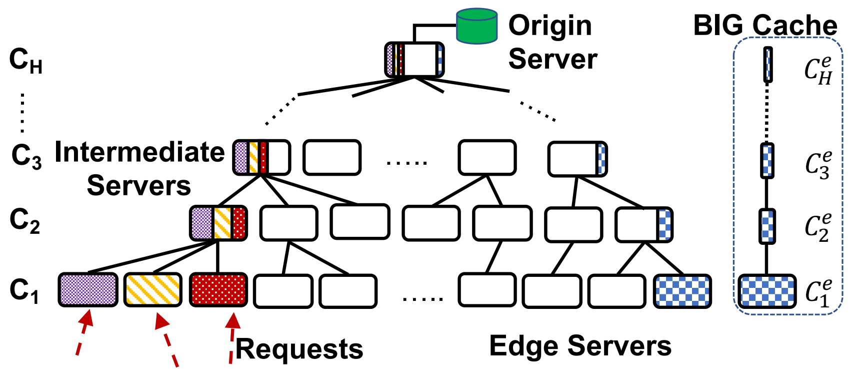


Figure 3.4: "BIG" Cache Abstraction

and not placed in the higher layer. For example, considering the simple LRU cache replacement policy being applied to a two-layer cache hierarchy, and assuming both layers are full. Upon receiving a request for an object cached at the second layer, it is moved to the first layer, and another object from the first layer is evicted to be inserted at the second layer; *i.e.*, only one copy is maintained throughout the hierarchy in the "BIG" cache abstraction, similar to the single cache case. Therefore, we can fully utilize the available cache resources at the intermediate cache servers.

3.4.1 Theoretical Performance Analysis

"BIG" cache abstraction has another advantage as it makes the theoretical performance analysis more manageable, because existing caching policies can be applied directly to the virtual "BIG" cache, as if it is a single cache. Using "BIG" cache, performance analysis of various caching policies for a single cache under IRM and renewal processes (such as [19]) can be directly applied to a tandem cache network. Given the analysis for a single cache under caching policy P, we show how it can be extended to calculate an approximate analysis for a line of caches using "BIG" cache abstraction.

Considering a network of $H+1$ layers, where C_{H+1} represents the origin server, which has a permanent copy of the object collection $\mathcal{O} = \{O_1, O_2, \dots, O_N\}$. User requests are first sent to edge server C_1, according to the request rate of each object λ_i, with a total request rate denoted by $\lambda = \sum_{i=1}^{N} \lambda_i$. The cache size of each layer is denoted by C_h,

$1 \leq h \leq H$. The size of the virtual "BIG" cache is denoted by $C_B = \sum_{h=1}^{H} C_h$. Thus, we can apply the same analysis for single cache by substituting the single cache size C with the virtual "BIG" cache size C_B. The intuition behind this approximation approach is to consider the cache network as a black box, which receives user requests with an aggregate rate λ, and has a total storage capacity C_B. Content objects can be stored at any layer of the cache hierarchy, which does not affect the overall hit probability p_i of object O_i. Thus, the percentage of requests satisfied by the cache network does not change depending on the location of the object in the hierarchy. The user latency is the metric which depends on which layer the object is served from.

Given a caching policy P, an aggregate request rate λ, for a cache size C, and the equations to calculate the hit probability of object O_i are defined by $p_i(C, \lambda, P)$[2] as a function of C, λ for a caching policy P. Using "BIG" cache abstraction, we can estimate the hit probability of each object at each layer in a line of caches. The hit probability of object O_i at layer $L1$ can be defined as $p_{i1}(C_1)$. Then, the hit probability of object O_i being served from either layer $L1$ or layer $L2$ can be calculated using the summation of the size of the caches at the first two layers. We introduce a new notation for this hit probability as $p_{i,[1:2]}(C_1 + C_2)$. Hence, the hit probability of object O_i being served from layer $L2$ can be calculated by $p_{i2} = p_{i,[1:2]} - p_{i1}$. In general, the hit probability of object i at layer L_h can be calculated by the difference between the hit probability of object O_i being served from the first h layers (*i.e.*, using cache size $C_{[1:h]} = \sum_{j=1}^{h} C_j$) and being served from the first $h-1$ layers using $C_{[1:h-1]}$. By iterating over all layers, starting from layer $L1$, we can calculate the hit probability for each layer as following:

$$p_{ih} = \begin{cases} p_{i,[1:1]} & \text{if } h = 1, \\ p_{i,[1:h]} - p_{i,[1:h-1]} & \text{if } 2 \leq h \leq H, \end{cases} \tag{3.1}$$

where $p_{i,[1:h]} = p_i(C_{[1:h]}, \lambda, P)$ and $C_{[1:h]} = \sum_{j=1}^{h} C_h$. Thus, we can calculate other performance metrics, such as average latency, origin server load, ... etc, using the object access information received at edge servers. Hence, we can avoid the analysis of the complex interactions between layers to estimate the object request arrival process for the next layer, which hinders the extension of the theoretical analysis from a single

[2]For simplification, the request rate λ and the caching policy P are omitted when the context is clear.

cache to a hierarchical cache network. For example, Markov processes are used for analyzing a tandem cache in [19]. However, this becomes challenging with larger values of H.

As an example, when LRU is applied independently for a tandem cache of two layers, using Che's approximation [21], and the analysis in [19], we can calculate the hit probability of object O_i at each layer as following:

$$\begin{aligned} p_{i1}^{(I)} &= 1 - e^{-\lambda_{i1} T_{C_1}} \\ p_{i2}^{(I)} &\approx 1 - e^{-\lambda_{i2}(T_{C_2} - T_{C_1})} \end{aligned} \tag{3.2}$$

where T_{C_h} is the characteristic time of the cache at layer h, and can be calculated using the cache size C_h, $\lambda_{i1} = \lambda_i$ is the arrival rate of requests for object O_i at cache layer $L1$, and $\lambda_{i2} = \lambda_{i1}(1 - p_{i1}^{(I)})$ is the estimated request arrival rate for object O_i at cache layer $L2$, which depends on the hit probability of object O_i at cache layer $L1$ (see [19] for more details).

In case of applying LRU for "BIG" cache, the hit probability of object O_i being served from any layer can be calculated by using the aggregate cache size ($C_1 + C_2 = 2C$). Using Equ. 3.1, the hit probability of object O_i at $L2$ can be calculated as following:

$$\begin{aligned} p_{i}^{(B)} &= 1 - e^{-\lambda_{i} T_{2C}} \\ p_{i2}^{(B)} &= p_{i}^{(B)} - p_{i1}^{(B)} \\ &= e^{-\lambda_i T_{C_1}} - \ e^{-\lambda_i T_{2C}} \end{aligned} \tag{3.3}$$

We can prove mathematically that $p_{i2}^{(B)} > p_{i2}^{(I)}$. Thus, using "BIG" cache allows better utilization for the second layer, and the overall cache performance. Through the remaining of the chapter, (I) is used to indicate applying caching policies at each layer independently, while (B) is used for applying caching policies as a single consistent strategy for "BIG" cache formed by the cache pieces from different layers. Fig. 3.5 shows the numeric results for comparing $p_{i2}^{(B)}$, $p_{i2}^{(I)}$ for $N = 100, H = 2, C_1 = C_2 = 10$, and requests following Zipf distribution with $\alpha = 1$. We can see that "BIG" cache better utilizes the cache resources leading to higher hit probability for all objects. Thus, better overall cache performance as we show via simulation next.

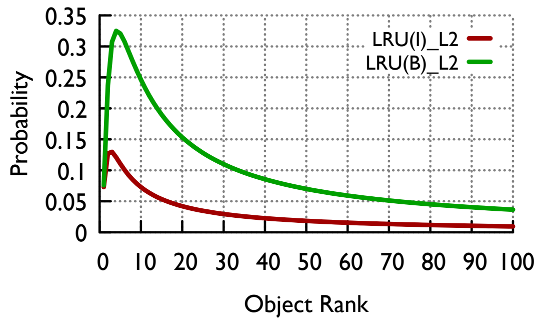


Figure 3.5: Object hit probabilities at the second-layer cache C_2 under LRU(I) and LRU(B): $N = 100$ and $C = 10$

3.5 dCLIMB: Caching Strategy for "BIG" Cache

In the previous section, we illustrated how "BIG" cache can help improve the performance of individual caches, as well as the overall performance of the cache network. However, applying some caching policies to "BIG" cache may incur some additional overhead to maintain their consistency. In the section, we explain how this additional overhead might occur, compare the performance of different caching strategies, and propose a caching strategy that overcomes this overhead.

For instance, LRU sorts cached objects according to their recent received request, such that the most recent requested object is at the head of the queue, and the least recent at the tail. Extending the same concept to "BIG" cache, the queue is now distributed among the cache layers. The head of the queue is at the edge server, and its tail is C_H (the layer before the origin server). Thus, if a requested object is cached at C_h, where $h \neq 1$ (*i.e.*, edge server), the object is moved to the head of the queue (at the edge server). When the cache is full, this triggers a series of object movements across boundaries of the distributed caches to maintain a consistent order of objects according to their recent request. Similar behavior applies for other cache replacement policies. Thus, in order for a caching policy to be beneficial, it should be efficient in terms of having a high hit probability, cache utilization, and also incurs low management overhead. This motivates us to design a caching mechanism for "BIG" cache, which minimizes object movements across the boundaries of distributed caches.

Considering all cache pieces of “BIG” cache are “glued” together to form a large single (global) *queue* data structure with C_1 as its head, C_H as its tail, and linked together by individual queues maintained at each cache server. Intuitively, the ideal goal of the caching mechanism is to have the object with the most frequent and recent access at the head of the queue, and the remaining objects are organized according to the order of their frequency and recent access, from high to low (see below for a more formal description). When a cached object receives a request, it is swapped with the object in the preceding index. If the requested object is not cached anywhere in the “BIG” cache, it is appended at the tail of the queue, *i.e.*, at the end of the queue maintained at the last server, C_H. We refer to this caching mechanism as ***dCLIMB***, which is a generalization of the *CLIMB* algorithm, first studied in [15] for a single cache, and we apply it to a set of distributed caches acting as one “BIG” cache. Under *dCLIMB*, each object access triggers at most one object swap operation. If the accessed object is currently at the head of the queue maintained at cache C_h, it would cause a swap operation across the cache boundaries with the cache C_{h-1}, if $h \neq 1$; otherwise, the swap operation is performed locally within the C_h cache.

The *dCLIMB* algorithm is formally described as following: assuming each cache C_h is organized as a queue of C slots (*i.e.*, cache size), and each slot is able to hold one object. Assuming object O_i is cached at slot m, $1 \leq m \leq C$, at layer C_h, $1 \leq h \leq H$. When a request for object O_i is received, there are three possible cases: i) if $m \neq 1$, then O_i is swapped with O_j at slot $m-1$ in C_h, ii) if $m = 1$ and $h = 1$, nothing happens, and iii) if $m = 1$ and $h = 2, 3, \ldots H$, O_i is swapped with O_j at slot C in C_{h-1}. In addition to the main cache at layer L_H, *dCLIMB* implements a temporary cache to avoid caching every requested object at the main cache C_H, which leads to the eviction of more popular objects. When an object is requested, only its meta-data is inserted in this temporary cache, and only objects at the head of the temporary cache are moved to the main cache of C_H.

The advantages of *dCLIMB* are multi-fold: It is a *self-adaptive* request-driven strategy, which decides automatically how objects should be placed along different cache layers, without the knowledge of user access patterns *a priori.* It attains higher hit probability, than the other classical cache replacement policies, by embedding the observed user access pattern for each object in its position in the queue. Moreover, *dCLIMB* is capable of dynamically adapting automatically to changes in user access patterns and flash crowds. For example, when an object receives a sudden burst of requests, *dCLIMB* would gradually move it to a lower cache layer closer to users, thus reducing its latency. However, this process may take some time, depending on how large the burst of requests is.

Finally, *dCLIMB* incurs minimal object movement overheads, compared to other caching policies. For example, for each request *dCLIMB* needs to update the indexes of only two objects; only if the object is at the head of its cache layer, it needs to be swapped with the lower layer. In contrast, LRU needs to maintain a queue of requested objects; each time an object is accessed, it is moved or inserted at the head of the queue, which requires more operations. LFU and k-HIT need to keep track of access counts (and sometimes timers), and require queue operations for object insertions or movements.

3.6 Cache Allotment for "BIG" Cache

One of the challenges of "BIG" cache abstraction is how to (logically) partition the cache resources at intermediate caching nodes to allot appropriate cache resources to form one (virtual) "BIG" cache with respect to each edge server. In this section, we illustrate how this allotment problem can be formalized as an optimization problem.

Using the same notations in §3.2.1 and §3.4. "BIG" cache is defined by the path β^e from an edge server C_e to the origin server C_o, passing by a sequence of intermediate caches. Hence, an intermediate cache C_h might be shared among several edge servers on their paths to the origin server. Hence, it needs to allot a piece of its cache to each edge server, C_h^e $(0 \leq C_h^e \leq C_h, e \in \mathcal{E})$, to form a "BIG" cache for edge server C_e. Let U^e be the performance objective function for "BIG" cache C^e. It is assumed to be strictly concave, increasing, and continuously differentiable. U^e can be a function of the overall hit rate, or the latency perceived by users. Thus, U^e is an implicit function of u_n^e, in which u_n^e represents the stationary occupancy of object O_n^e at "BIG" cache C^e.

$$\begin{aligned}
\underset{\substack{u_n^e \in [0,1] \\ C_h^e, C^e > 0}}{\text{maximize}} \quad & \sum_{\{e|C_e \in \mathcal{E}\}} \sum_{n=1}^{N^e} U^e(u_n^e) && \text{(3.4a)} \\
\text{s.t.} \quad & \sum_{n=1}^{N^e} u_n^e \leq C^e, \quad \{e|C_e \in \mathcal{E}\} && \text{(3.4b)} \\
& C^e = \sum_{\{h|C_h \in R^e\}} C_h^e, \quad \{e|C_e \in \mathcal{E}\}, && \text{(3.4c)} \\
& \sum_{\{e|C_e \in \mathcal{E}\}} C_h^e \leq C_h, \quad h \in \mathcal{H} && \text{(3.4d)}
\end{aligned}$$

We aim to maximize the performance of each "BIG" cache C^e under the following constraints. The summation of the occupancy probability of all objects cached in each "BIG" cache should not exceed its capacity, Equ. 3.4b. The capacity of each "BIG" cache is the summation of each cache piece allotted to it from different intermediate servers along the path $\mathcal{R}^e$, Equ. 3.4c. Finally, the total capacity allotted to different "BIG" caches sharing an intermediate cache C_h at layer h should not exceed the capacity of C_h, Equ. 3.4d.

The utility maximization formulation of the optimization problem can be decomposed into two sub-optimization problems: 1) the optimal *cache allotment* problem which determines the optimal cache capacity for each "BIG" cache to maximize the performance for the given request arrival process at each edge server, and 2) the optimal *object placement* problem which determines the optimal stationary occupancy probabilities for objects in each "BIG" cache with a given cache capacity to meet user QoE expectations. In Equ. 3.4, the variables C_h^e & C^e represent the solution of the allotment problem, and u_n^e represents the solution of the object placement problem. In [25], we show how the problems of cache allotment and object placement can be decoupled using an *optimization decomposition framework* for cache network management, where we develop a primal-dual algorithm to solve the cache allotment problem and the object placement problem separately and iteratively.

It is important to clarify that partitioning the available cache space does not mean that multiple copies of the same object are cached, when multiple edge servers share the same cache server. Only one physical copy is cached among different edge servers sharing the same physical intermediate cache. This leaves more space to cache other objects from higher layers, which helps improve the hit probability, and also user latency. Cache partitioning is only virtual (logical), however, it is possible to keep track of the statistics of each "BIG" cache separately, where objects in each "BIG" cache are managed independently of each other. This independent management of objects does not impact the performance of the overall system, as shown by the results in §3.7. Moreover, intermediate servers can also be edge cache servers for other user populace. In such cases, these servers receive requests coming directly from users, in addition to other requests received from edge servers. These requests may vary in the objects of interest, and also the number of requests. This variation can be handled by the cache allotment (*i.e.*, partitioning), which considers all possible streams of requests arriving at the current server. A similar approach to the technique in [26] can be applied.

3.7 Evaluation

In this section, we use simulation to further show the advantages and benefits of our proposed notion "BIG" cache compared to the existing caching policies. We compare the performance of individual cache layers, and also the overall performance and utilization of the cache network. We conduct experiments for a tandem cache network, a hierarchical tree cache network, and compare the different allotment strategies for "BIG" cache.

3.7.1 Tandem Cache Network

Using a tandem cache network with four layers ($H = 4$), the size of each is $C_h = 10, 1 \leq h \leq H$. The origin server lies at $L5$, which has a permanent copy of the object collection of $N = 100$ objects. User requests are sent to the edge server at $L1$, if the requested object is cached, a copy is returned to the user. Otherwise, the request is forwarded along the path to the origin server till a copy is found. When the object is found at cache C_h, we simulate various object allocation strategies to decide which layer should cache a copy of the requested object. 1) leave-copy-everywhere (LCE): each layer on the way back caches a copy of the object, 2) edge-caching-only (OE): the edge server only

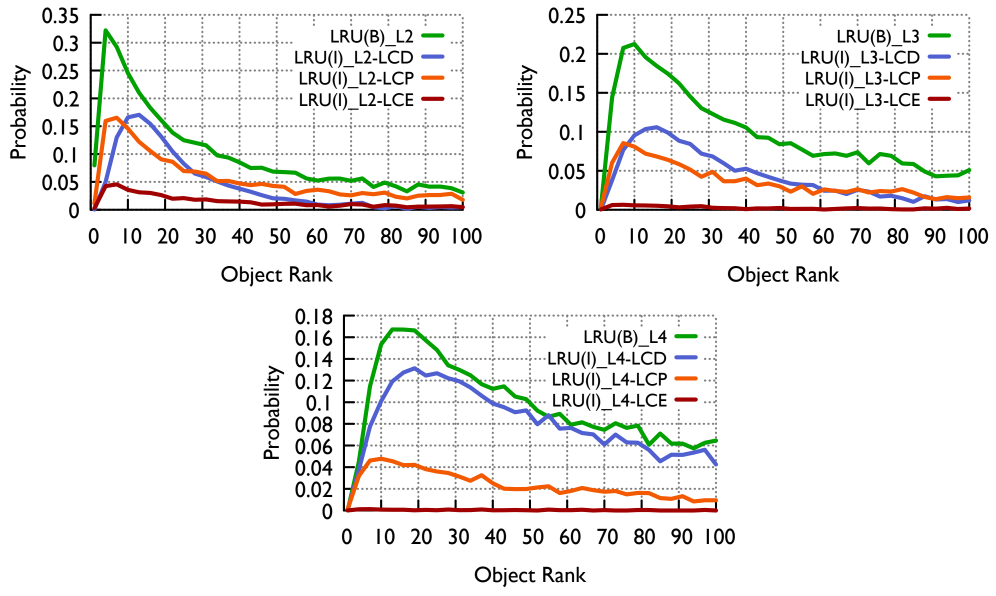


Figure 3.6: LRU - Object hit probabilities, $N = 100$, $C = 10$, $H = 4$, $R = 1M$, $q = 0.5$ for LCP

caches a copy, 3) leave-copy-probabilistically (LCP): each layer caches a copy according to a probability q (in our experiments we use $q = 0.5$), and finally 4) leave-copy-down (LCD): only the lower layer caches a copy (C_{h-1}). We generate $R = 1M$ requests following the popularity of objects according to Zipf distribution with $\alpha = 1$. We use LRU as the cache replacement policy, being applied independently in each layer, along with these object allocation strategies. We compare these techniques with LRU being implemented as a single consistent strategy for the "BIG" cache of the tandem cache network.

Fig. 3.6 shows the object hit probability $p_{ih}, 2 \leq h \leq H$, at the intermediate layers $L2$, $L3$, and $L4$, which represents the percentage of requests being served from each layer for each content object. For LRU(I)-LCE, we can notice that there is no much improvement in the hit probability for these layers. $L3$ & $L4$ are nearly not being utilized, and layer $L2$ even have a worse performance, compared to Fig. 3.5, when the value of H increased, due to cascaded thrashing problem (see §3.2.2 for more details). On the other side, we can notice that the performance of LRU(B) for $L2$ is not affected by increasing the number of layers. Moreover, it has a higher object hit probability,

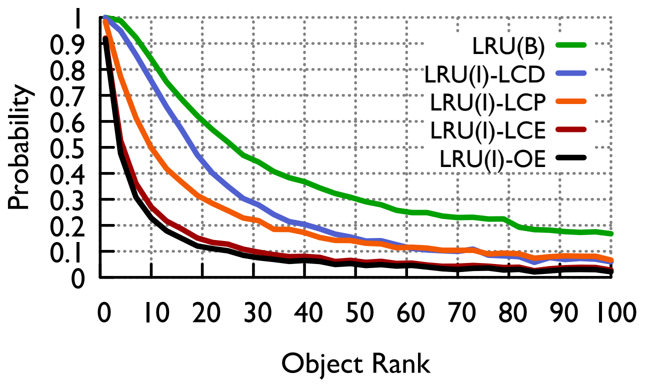


Figure 3.7: Overall object hit probability

especially for $L3$ and $L4$. Using LCP and LCD for object allocation, slightly improve the performance of these intermediate layers. However, they still cache at least one more copy in the cache network. Thus, "BIG" cache still outperforms these strategies. Only for $L4$, the performance of LRU(I)-LCD and LRU(B) is close, due to having the same behavior at this layer, by caching objects retrieved form the origin server. All strategies have the same performance at $L1$.

Next, we examine the effect of "BIG" cache on the overall performance of the cache network, after examining each layer individually. Fig. 3.7 shows the aggregate hit probability for each object being served from any cache layer $p_i = \sum_{h=1}^{H} p_{ih}$. We can notice that "BIG" cache is outperforming the other techniques, while LRU(I)-OE has the worst performance due to only utilizing edge caches. By aggregating the hit probability of each object being served from a specific cache, we can get the overall cache utilization $\Gamma_h = \sum_{i=1}^{N} p_{ih} \leq C_h$. A caching strategy fully utilizes the cache when $\Gamma_h = C_h$. Fig. 3.8 shows the cache utilization for each layer. We can notice that only "BIG" cache is able to fully utilize all cache resources at each layer. While, LRU(I)-LCE can barely utilize 20% of the cache at $L2$, and almost 0% of the higher layers. Using LRU(I)-LCD and LRU(I)-LCP improve the utilization of the caches at higher layers slightly, but still they are under-utilized. Different replication strategies can only fully utilize the edge server.

We also tried a more sophisticated caching policy K-Hit, by which each cache server only caches a copy of the content after receiving K requests within a pre-specified threshold. We use $K = 4$ for our simulation, we compare K-Hit being implemented

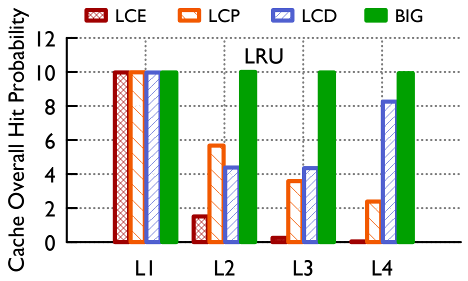


Figure 3.8: Overall cache hit probability

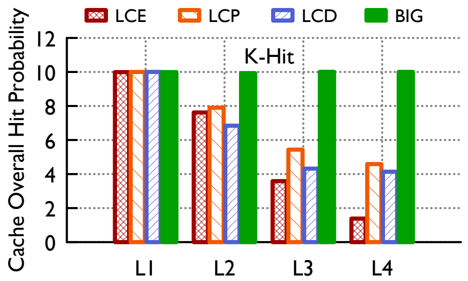


Figure 3.9: Overall cache hit probability

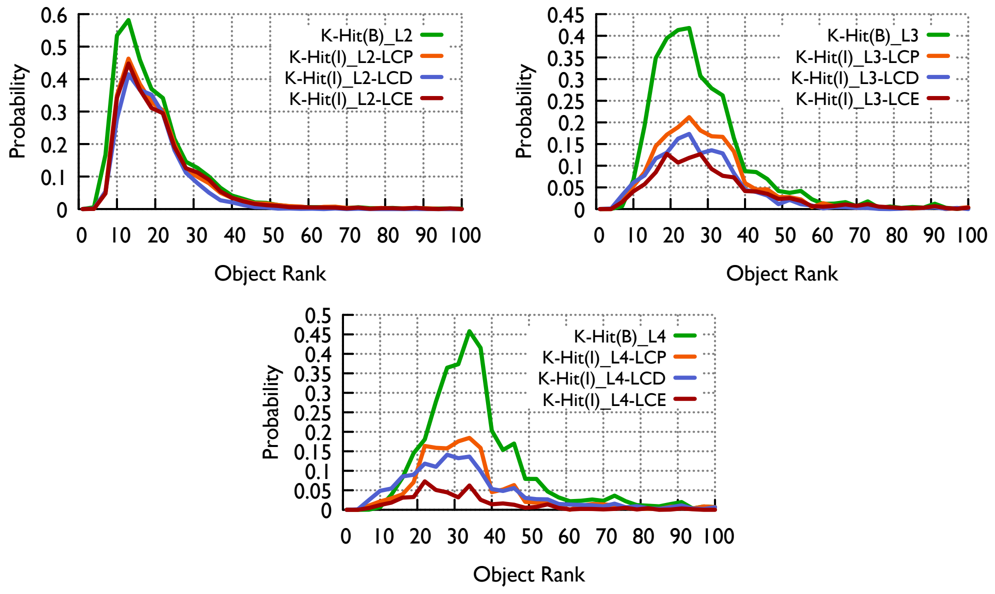


Figure 3.10: k-HIT - Object hit probabilities: $N = 100$, $C = 10$, $H = 4$, $R = 1M$, $k = 4$, $timer = 500\ hits$, $q = 0.5$ for LCP

independently at each layer K-Hit(I) vs. implementing it for "BIG" cache K-Hit(B). The performance of the independent caching policy has improved, but our previous observations still hold. Fig. 3.10 shows the object hit probability p_{ih} at the intermediate layers $L2$, $L3$, and $L4$. We can still notice that K-Hit(B) is outperforming the other caching strategies, and have around 50% increase in hit probability of popular objects in higher layers. The performance of K-Hit(I)-LCE, K-Hit(I)-LCD, and K-Hit(I)-LCP have improved in general, but still they have degraded performance for layers $L3$ and

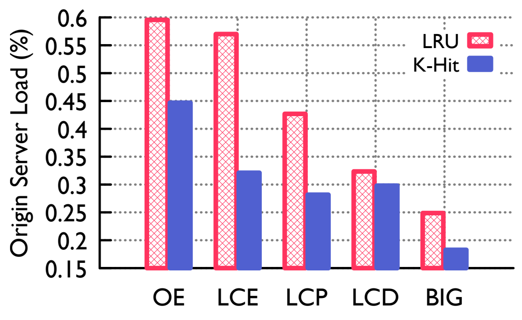


Figure 3.11: Percentage of Requests served by Origin Server

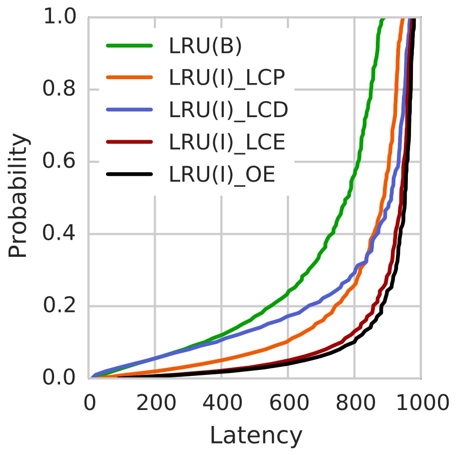


Figure 3.12: LRU Latency

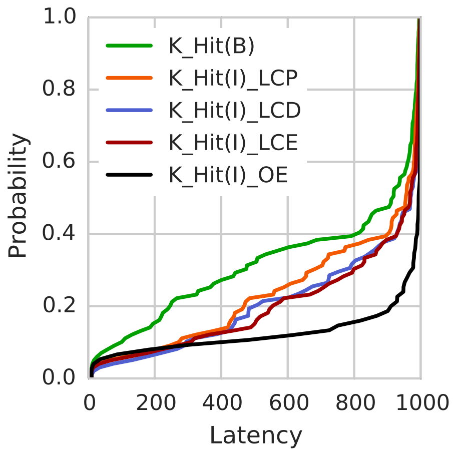


Figure 3.13: K-HIT Latency

$L4$. Fig. 3.9 shows the cache utilization Γ_h for K-Hit, we can notice that "BIG" cache efficiently utilizes all cache resources for all layers, while the utilization of the other cache allocation strategies decreases as we approach the origin server.

Consequently, we can see in Fig. 3.11 that "BIG" cache minimizes the percentage of requests being served from the origin server, $\Omega = 1 - \sum_{i=1}^{N} \sum_{h=1}^{H} p_{ih}$, for both LRU and K-Hit. Taking the user perspective of the cache network into consideration, Figs 3.12, 3.13 show the average estimated latency to fetch each object. We use $(\phi_1 = 1, \phi_2 = 10, \phi_3 = 50, \phi_4 = 100, \phi_5 = 500)$ for the latency of retrieving an object from each layer. We clearly find "BIG" cache incurs the minimum latency outperforming the other methods.

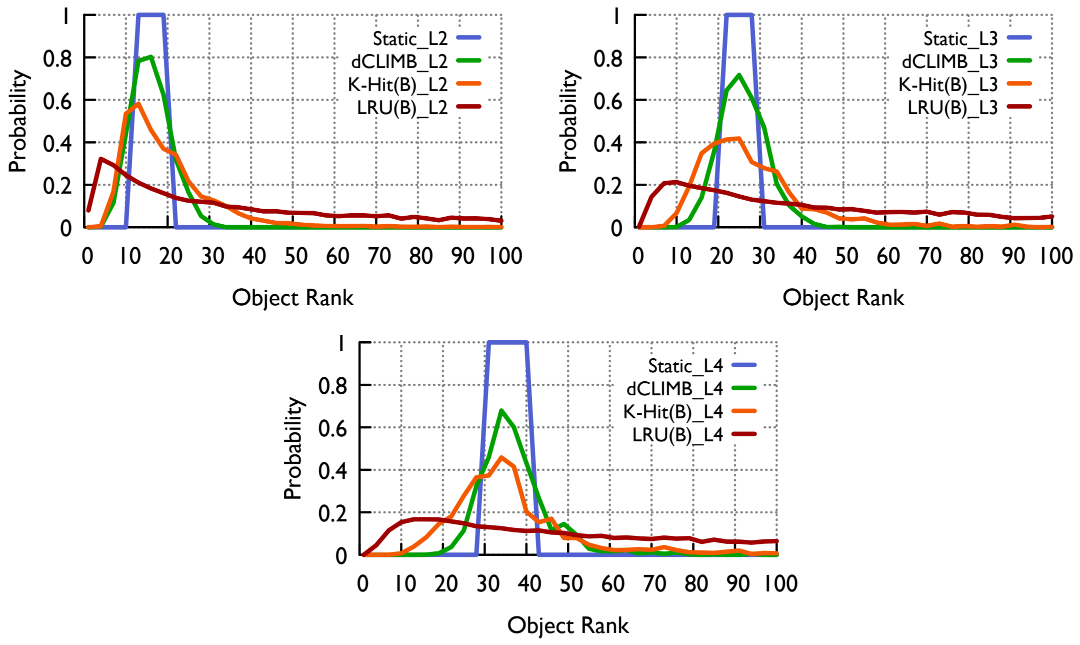


Figure 3.14: Object hit probabilities, $N = 100$, $C = 10$, $H = 4$, $R = 1M$, $k = 4$, $timer = 500\ hits$

Fig. 3.14 shows the performance of LRU, k-HIT, and *dCLIMB* applied to "BIG" cache compared to static caching, we can notice that *dCLIMB* actually utilizes the intermediate layers of the cache hierarchy with a performance close to static caching, which we consider as the optimal policy, if the user access patterns are known a priori. However, *dCLIMB* achieves this performance without the knowledge of user access patterns a priori, leading to the most popular content objects being cached at edge servers, and less popular objects at higher layers in the hierarchy. The number of insertion and eviction operations for all cache layers approximately were: LRU: 2.7M, K-HIT: 222K, *dCLIMB*: 109K. We can notice that *dCLIMB* nearly requires half the number of operations required by k-HIT, and at the same time it provides the closest performance to static caching, and increases the object hit probabilities at different caches, without the need to keep track of counters and timers. Thus, *dCLIMB* minimizes the problem of thrashing (which produces wasteful disk operations that consume a lot of energy), and creates significant energy savings at the cache servers. Therefore, *dCLIMB* is a distributed and coordinated caching mechanism, with minimal complexity, and without the need for a global central controller.

3.7.2 Hierarchical Tree Cache Network

We use a topology following the multi-layered architecture of YouTube video delivery system revealed in [4]. In this architecture, cache servers are organized in a 3-tier cache hierarchy (primary, secondary, and tertiary servers). The tertiary servers are connected to the origin server. User requests are first sent to the primary (or edge) cache servers, which sends the requested objects back, if they are cached. Otherwise, the request is forwarded to the secondary layer. This process continues till the object is found. If none of the cache servers have the object, the request is forwarded to the origin server. With this notion, we consider a similar topology in the form of a binary tree, in which the root is the origin server (O), and the leaf nodes are the primary (*i.e.*, $L1$) servers, which receive user requests. The intermediate servers comprises of the secondary ($L2$) and tertiary ($L3$) servers, thus $H = 3$. We consider a collection of 10K objects of unit size, whose access probabilities follow a Zipf distribution with $\alpha = 1$. Two million requests are generated for each edge server. We assume the rate of requests to be uniform across all primary servers. Cache sizes of primary, secondary, and tertiary servers are set to be 1K, 2K, and 4K, respectively. Due to this uniformity, the intermediate cache servers are partitioned equally among the different edge servers sharing them (*i.e.*, the total size of each big cache is 3K, that's 1K for each cache layer).

We compare the performance of applying LRU in "BIG" cache LRU(B) and applying LRU independently LRU(I) in each cache server. We also applied different object allocation methods (LCE, LCD, LCP) when LRU is applied independently. Fig. 3.15 shows the object hit probability at the different layers and the origin server. Although not shown, we find that LRU(I) and LRU(B) served almost a similar percentage at edge servers. Fig. 3.15 shows LRU(B) served a higher percentage of requests for almost all objects in the intermediate layers ($L2$ and $L3$). Moreover, LRU(B) has the least percentage of requests being served by the origin server. All of this suggests that "BIG" cache abstraction leads more objects to be served from the intermediate layers than going all the way to the origin server.

We further compare the overall object hit probability from cache servers at layers $L1$, $L2$, and $L3$ in Fig. 3.16, and the average estimated latency to fetch each object in

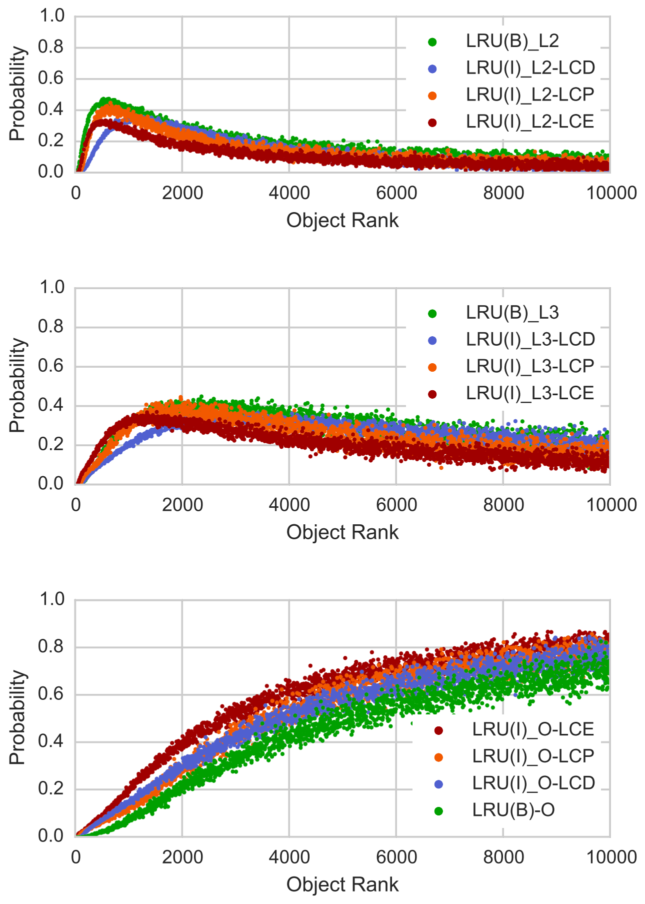


Figure 3.15: Object hit probability at layers ($L2$), ($L3$) & origin (O)

Fig. 3.17. We clearly find that "BIG" cache abstraction outperforms all other object allocation methods. The main reason is that by caching only one copy at each path from the edge server to the origin server, we allow space for more objects to be cached at lower layers, and thereby increase their hit probability, and at the same time minimize the origin server load. Moreover, in "BIG" cache abstraction, objects evicted from one layer are not totally discarded, but they are cached in the higher layers. Thus, "BIG" cache abstraction enables us to efficiently utilize the available cache resources along the path to the origin server, while taking into consideration the requests from all edge servers sharing the same cache server. These results also emphasize the previous conclusions and observations in §3.4. It is worth mentioning that the improvement in

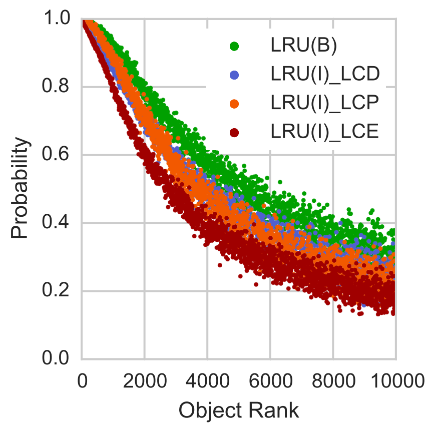


Figure 3.16: Overall Object hit probability

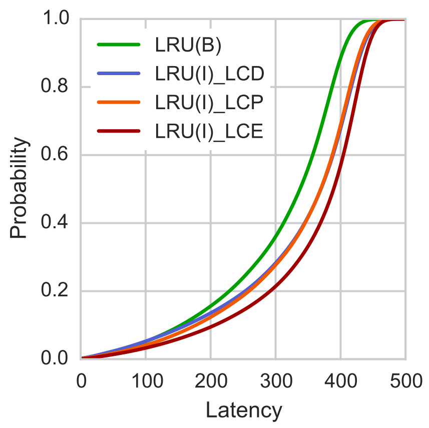


Figure 3.17: Overall Latency (CDF)

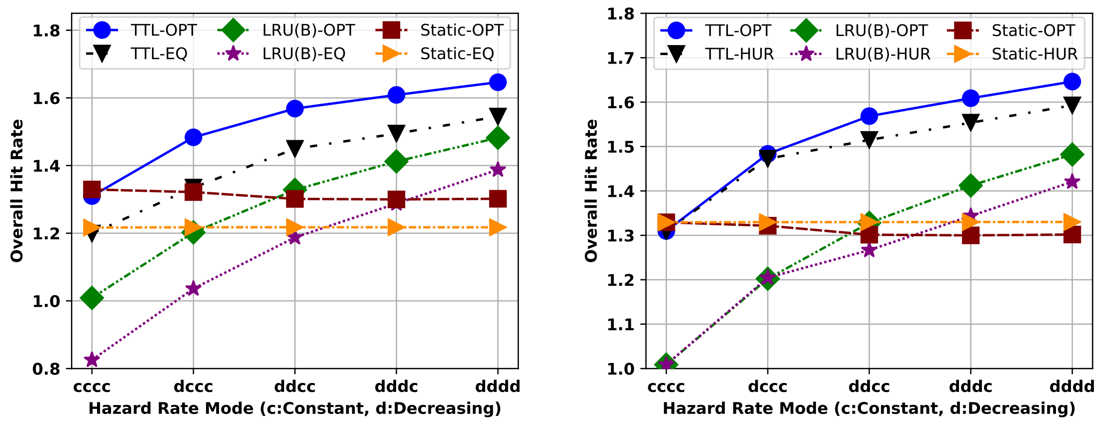
the performance, shown in this section, is just by using the simple LRU caching policy. We expect the performance to be even better using more sophisticated caching policies.

3.7.3 Comparing Cache Allotment Strategies

We use a hierarchical cache network topology in a form of a binary tree with 3 layers. The root of this tree is connected to an origin server, which has a permanent copy of all objects. There are four edge servers at layer $L1$, and three intermediate servers located at layers $L2$ & $L3$, resulting in four "BIG" caches corresponding to each edge server denoted by $B = \{B1, B2, B3, B4\}$. Cache servers at the same layer have the same capacity. Thus, the cache network capacity is represented by the total capacity of all caches in each layer $C = \{C_1, C_2, C_3\}$. Initially, $C_2 = 2 * C_1$ and $C_3 = 4 * C_1$. Requests characteristics at each edge server are independent of other edge servers, as well as the object catalogs. Each user populace attached to an edge server is interested in a set of objects represented by $N = \{N^1, N^2, N^3, N^4\}$. Initially, we assume the objects requested at each edge server are distinct and not common. The popularity of content objects in each catalog follows Zipf distribution with parameters $\alpha = \{\alpha^1, \alpha^2, \alpha^3, \alpha^4\}$. The aggregate request rate at each edge server is represented by $\lambda = \{\lambda^1, \lambda^2, \lambda^3, \lambda^4\}$. The monotonicity of the hazard rate function of request interarrival time distribution at each edge server is either constant denoted by (c), *e.g.*, Poisson distribution, or decreasing denoted by (d), *e.g.*, Pareto distribution.

The characteristic of object catalogs are represented by $\alpha = \{0.2, 0.2, 0.5, 0.5\}$, $N = \{300, 500, 1000, 1000\}$. The cache capacity is $\{C_1 = 30, C_2 = 60, C_3 = 120\}$. We use optimal TTL-caching, LRU, and static caching policies applied to the four "BIG" caches to manage the eviction of objects. In TTL-caching, upon caching object O_n^e at "BIG" cache C^e, a timer value T_n^e is set up. Any request received before the expiration of the timer T_n^e for object O_n^e results in a cache hit, cached objects are evicted upon the expiration of their timer. However, we use a practical TTL-caching in which expired objects stay in the cache, and are only evicted upon receiving a new object and the cache is full. Then, the object with the least recent expired time is removed. Ferragut *et al.* [27] show that practical TTL-caching has a close-to-optimal performance. LRU and static caching are two special cases of TTL-caching, where in LRU all objects have the same timer, and in static caching the timer assigned to each object is ∞. We consider the origin server load as the utility function for the cache network optimization defined in Equ. 3.4. We use $\lambda = \{1, 1, 1, 1\}$ and generate $10M$ requests for the objects of each edge server according to the monotonicity of the hazard rate function of request interarrival time distribution as mentioned above. We change the request interarrival time distribution for our experiments according to the following settings for the four edge servers $\{cccc, dccc, ddcc, dddc, dddd\}$.

By adopting "BIG" cache idea, we need to partition the intermediate caches to form one "BIG" cache for each edge server. Among the allotment strategies are: *1) Equal Allotment.* It is a simple approach which partitions each intermediate cache equally among the "BIG" caches sharing it. However, equal allotment does not consider the characteristics of the object requests among the different edge servers. *2) Heuristic Allotment.* This heuristic is based on *static caching.* The authors in [28] prove that allocating the top C most popular objects to a cache of size C leads to the highest cache hit rate. Since, "BIG" cache allows caching mechanisms to be used as a single caching policy for hierarchical cache. Thus, we can use the same fact for the heuristic as following: at edge servers, we allot the top C_1^e objects, based on the request pattern for this edge server. These objects are not considered for higher layers. Then, for each intermediate server C_h, we aggregate the received request streams for all edge servers passing through this server on their path to the origin server (*i.e.*, $C_h \in R^e$). We merge



(a) Optimal vs. Equal (b) Optimal vs. Heuristic

Figure 3.18: Overall Hit Rate: Optimal Allotment vs. Equal and Heuristic Allotments

and sort the objects of these streams[3], and allot the cache to C_h objects with the highest request rates. The number of objects from each edge server C_e that is among these top C_h objects, would be the size of cache piece allotted to C_e. *3) Optimal Allotment.* The optimal solution is obtained using the primal-dual algorithm to solve the cache allotment problem and the object placement for the optimization problem defined in Equ. 3.4 as detailed in [25].

Optimal vs. Equal. Fig. 3.18a shows that the optimal allotment yields a higher overall hit rate for all caching strategies (TTL, LRU(B), Static), LRU(B) refers to LRU applied to "BIG" cache. This is because optimal allotment considers the characteristics of object requests of the different edge servers, compared to equal allotment which treats them as if they are similar.

Optimal vs. Heuristic. In Fig. 3.18b, we can notice that for TTL & LRU(B) both allotments yield the same overall cache hit rate in the cases of $\{cccc, dccc\}$, where the constant hazard rate is the most dominant, but for the other cases the optimal allotment is better. As discussed in [27], static caching is optimal for constant hazard rate only. Thus, the heuristic approach is not suitable for decreasing hazard rate as it is based on static caching as described above. Therefore, when we change the request

[3]In this heuristic, we treat objects/object requests from different edge servers C_e as *distinct*, even though some are for the same objects.

interarrival time distribution, the heuristic allotment does not change, while the optimal allotment does. Static caching performs poorly even with the optimal allotment, because it considers the decreasing hazard rate, which is not accounted for in static caching as objects are always cached instead of probabilistic caching. Thus, it yields slightly lower hit rate when more edge servers have decreasing hazard rate.

Caching Policies Performance. From Fig. 3.18a & Fig. 3.18b, we can notice that for TTL & LRU(B) the hit rate increases as more branches receive requests with decreasing hazard rate, which leads to receiving more requests for the same object in a shorter duration compared to constant hazard rate, which is not the case for Static caching. Ferragut *et al.* [27] discussed the impact of hazard rate on LRU and static caching performance which matches our results as well.

3.8 Discussion

In this section, we discuss how we can relax the assumptions in our "BIG" cache abstraction.

Non-unit Object Size. We modify the cache size constraint Equ. 3.4b. Consider objects in catalog N^e to have sizes of $\mathcal{S}^e = \{s_1^e, \cdots, s_{N^e}^e\}$, then the constraint is redefined as:

$$\textstyle\sum_{n=1}^{N^e} s_n^e * u_n^e \leq C^e, \quad \{e|C_e \in \mathcal{E}\}$$

Multiple Origin Servers. We assume having multiple origin servers with full content replica, and each intermediate server contacts the closest origin server. In case the object catalog is divided among the origin servers, then we can consider the origin server to be logically centralized. This means each intermediate server still contacts the closest origin server, and the origin servers run a distributed algorithm to keep track of objects and exchange them.

User Pattern Changes. Another interesting design aspect for a caching mechanism is to be able to quickly adapt to changes in user access patterns. Especially, when unpopular objects receive a burst of requests and need to be cached at lower layers close to end users. Static caching is not capable of handling such scenarios. As we mentioned, our proposed *dCLIMB* is a self-adaptive request-driven caching mechanism, but it may take time to cache the objects at lower layers, as this depends on the length of the burst of requests. Thus, we can enhance *dCLIMB* by defining a step value, such that this step is used while swapping objects, and it can be defined according to the number of hits received for this object. For example, when a request is received for object O_i cached at position m, and has number of hits ht. It is swapped with the object O_j at position $m - Fn(ht)$, where $Fn(ht)$ is a function of the number of hits, which could be either polynomial or exponential. Thus, upon receiving more requests, objects can be cached faster at lower layers.

Unknown Object Access Probability. We assume that object access probabilities are known a priori. Although, in real systems extracting the access rate for each object is a challenging task. Hence, some techniques were developed recently to address this issue. Dehghan *et al.* [29] propose an estimation technique to approximately calculate the object access rate using TTL timers. This continuous estimation captures the changes in the request access patterns, which is then fed to the optimization framework. Another recent trend is to estimate object access rate using machine learning techniques (*e.g.*, [30, 31, 32, 33]). This is the focus of Chapter 4 which uses a deep learning based object access probability prediction.

3.9 Summary

We have made a strong case for "BIG" cache abstraction to effectively utilize the distributed storage of all cache servers in a cache network specially the intermediate cache servers. Through examples and simulations, we demonstrated that "BIG" cache abstraction can indeed eliminate the problem of (cascade) thrashing when cache servers operate independently with their own cache replacement policies. "BIG" cache abstraction significantly improves the overall performance of a cache network while also drastically reducing the loads and other performance constraints at origin content servers. "BIG" cache abstraction also opens up a number of new and challenging research questions and directions. As an initial step towards addressing some of these issues, we have developed the *dCLIMB* cache mechanism for “BIG” cache to minimize the overheads of moving objects across distributed cache boundaries. We also outlined an optimization problem formulation to address the cache allotment problem in the design of “BIG” cache abstraction.

Chapter 4

DeepCache: A Deep Learning Based Framework for Content Caching

4.1 Introduction

One of the most important decisions for content distribution networks (CDNs) is which object to cache and evict, given the limited capacity of the cache network and large number of objects due to the continuous growth of available online content and streaming services. Generally, caching policies can be classified either as reactive or proactive policies based on which entity controls the caching decision, and the available information to make these decisions. In reactive caching (such as Least Recently Used (LRU), Least Frequently Used (LFU), and their variants), individual cache servers decide which objects to cache purely based on the recent locally observed object access patterns. They are easy to implement and widely used in today's CDNs [34]. On the other hand, in proactive caching (such as static caching), centralized controllers have global view of user demands and object access patterns and decide which objects to cache, then push these objects to different cache servers. Reactive caching reacts faster to changes in object access patterns, but leads to caching non-popular objects, which are evicted before receiving their next request due to the lack of knowledge about future object popularity.

This leads to thrashing problem and wasting cache resources (as discussed in §3.2.2). Proactive caching is the optimal solution only if the object access pattern is stationary in which the most popular objects are placed closer to end-users. Hence, it cannot cope with sudden changes in object popularity as reactive caching.

Content objects are heterogeneous as they vary in size (*e.g.*, web pages vs. videos), access pattern, and popularity. A study in [35] shows that 70% of objects served by a cache server are requested only once over a period of days. Object access patterns are frequently changing due to the frequent changes in object popularity as shown by the study in [36] using real traces, object popularity changes within each day according to the diurnal pattern, and also over days according to the object's life span. In addition, changes in request routing algorithms due to network/server failures can also cause changes in object access patterns. Due to these frequent changes, the assumption of stationary object access patterns becomes invalid. Thus, caching algorithms can not rely on the locally observed object access patterns for making decisions. On the other hand, manually tuning the caching algorithm for each cache server according to the changes of request access patterns is very expensive and is not scalable. Hence, our goal is to develop a self-adaptive caching framework, which automatically learns the changes in request traffic patterns, especially bursty and non-stationary traffic, and predicts future content popularity, then decides which objects to cache and evict accordingly to maximize the cache hit.

In recent years, recurrent neural networks (RNN) have become the cornerstone for sequence prediction. RNNs have shown their unchallenged dominance in the area of natural language processing [37], machine language translation [38], speech recognition [39], and image captioning [40]. Many variants of RNN exist in literature, among which Long Short-Term Memory (LSTM) [41], Gated Recurrent Unit (GRU) [42] are the most popular ones for sequence prediction. Thus, it is natural to investigate their ability to predict content popularity where content requests arrive in a form of a sequence.

In this Chapter, we present our DeepCache framework, which successfully demonstrates the ability of our LSTM based models to predict the popularity of content objects. The main contributions of this work are two folds. We recognize the problem of content popularity prediction as a seq2seq modeling problem, and to our knowledge, we are the first to propose LSTM Encoder-Decoder model for content popularity prediction. Secondly, we create DeepCache framework for making end-to-end caching decisions based on the predicted popularity. In the following sections, we discuss existing caching mechanisms, related work and challenges in applying machine learning to the content caching problem in §4.2. We illustrate our DeepCache Framework in §4.3. We present our evaluation for DeepCache framework by applying it to existing caching policies like LRU and K-LRU, and show that it *significantly* boosts the number of cache hits in §4.4. We provide insights on expanding DeepCache in §4.5, and summarize the Chapter in §4.6.

4.2 Related Work

The main issues with the existing caching policies is that they depend on the history of the received requests for specifying object popularity and hence deciding which objects to cache and evict regardless of the changes which might happen in the future. Hence, the decisions made to cache some objects might lead to thrashing by caching objects which might not be popular, and evicting more popular objects as the cache capacity is limited. For example, LRU, LFU, and their variations decide the order of cached objects for replacement based on locally observed access patterns. Another example is timer-based caching which sets a time-to-live (TTL) timer when the object is caches, and when the timer expires the object is evicted from the cache. According to Che's approximation [21], objects within a cache can be viewed independently, and hence their hit probability can be calculated individually. Hence, Ferragut *et al.* in [27] formulated the timer set up as an optimization problem to maximize cache hit rate given the knowledge of object inter-arrival process and popularity distributions. However, the extracting the object characteristics in real systems is a very challenging task. To account for object heterogeneity, burstiness, and non-stationary nature of real-word content requests workloads, Basu *et al.* designed an adaptive algorithm to set up TTL-values for objects in [35], however their algorithm also relies on the history of the received requests to change the TTL-values without any consideration for any changes which may happen in the future as all the reactive caching algorithms.

In order to learn the object characteristics and predict the future characteristics, we need to utilize machine learning techniques. Among the recent efforts to learn object access patterns, in [43] Hashemi *et al.* employed LSTM to prefetch program counter's memory address to avoid on-chip misses by treating the problem as a sequence classification. In contract, in our proposed DeepCache framework, we employ LSTM Encoder-Decoder Model for object popularity prediction and treat the prediction problem as a seq2seq modeling. Pensieve [30] is the most closest work that partially focuses on object popularity prediction, however their object prediction is based on statistics accumulated over time regardless of the temporal behavior which we account for in our prediction approach. Another machine learning technique which is reinforcement learning (RL) has also been introduced in [30, 31] recently for designing caching policies based on local and global object popularity. This work can be integrated into our caching policy module to make better decisions, and hence complements our DeepCache framework.

Challenges: The optimal caching policy is to have every request result in a cache hit, however this is not possible due to the limited cache capacity. Hence, to improve the cache efficiency we need to predict with a high degree of accuracy object characteristics ahead of time to be able to cache these objects before the arrival of their requests. Object characteristics include object popularity, user request patterns, life-spans, ... etc. This is very challenging due to changes in content request patterns, request burstiness, and non-stationary nature of real-world content object requests. Even using machine learning techniques is not straight-forward due to the following challenges: 1) how should the caching problem be modeled for machine learning?, 2) which object characteristics should be used as input features to the machine learning models given their diversity and heterogeneity, and 3) what actions should be taken based on the machine learning output and when to take these actions to update the cache? Therefore, in this chapter we try to answer the following question: *can we develop a self-adaptive and machine learning driven caching mechanism that is able to generalize to different and time-varying content object characteristics (e.g., arrival patterns, popularities, life-spans) and improve cache efficiency?*

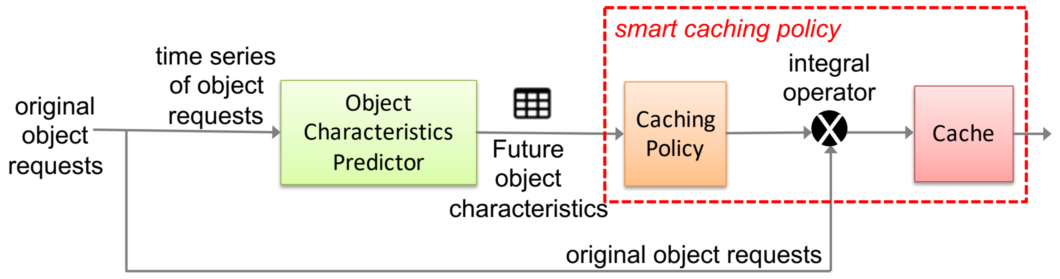


Figure 4.1: Data Flow in DeepCache

4.3 DeepCache Framework

4.3.1 Overview

As mentioned in the previous section that caching policies depend on the past which might lead to *cache thrashing* as mentioned in §3.2.2. Hence, the goal of our proposed DeepCache framework is to use the state-of-the-art machine learning (ML) algorithms to predict future characteristics of objects to be available for caching policies to *proactively* decide which objects to cache and evict. For example, knowing what objects are popular in the future, the caching policy can prefetch them ahead of time if they are not cached before their requests arrive. Also, when the cache is full less popular ones can be evicted instead of evicting objects which will be requested soon. Thus, DeepCache can increase the cache hit efficiency and utilization, reduce user latency, and minimize/eliminate thrashing problem. Additionally, if objects have different revenue for cache hits, then the caching policy can also use the prediction to decide which objects to cache to increase the profit.

Fig. 4.1 shows the data flow in DeepCache, in which the input to the ML prediction model is a timer series of the objects requests received at the cache, and the output is the future object characteristics. To give the caching policy more flexibility to decide which objects to cache and evict, the predicted object characteristics are for immediate future (next 1-3 hours), near future (next 12-14 hours), and far future (next 24-26 hours). DeepCache framework can also operate with traditional existing caching policies through the integral operator, which combines the information from the original object requests and the output of the caching policy logic to be sent to the cache.

Hence, this architecture allow for a better, novel, and *"smart"* caching policies which rely on the predicted object characteristics instead of just the history. In §4.4, we show how our DeepCache can be combined with traditional LRU-based caching policies to improve cache efficiency.

Sequence-to-sequence learning (seq2seq) [44] is a machine learning technique to train models for converting sequences from one domain (input) to sequences in another domain (output). It has shown its dominance in the areas of natural language processing, machine translation, and speech recognition. The flexibility of seq2seq modeling comes from the ability of having different input/output sequence length, and its ability to predict variety of outputs. Hence, we raise the question about its usage for predicting object characteristics such as: 1) predicting object's popularity over multiple time steps in the future, 2) predicting sequential patterns in object requests for related-objects, 3) identifying anomalies in object requests such as flash crowd phenomenon by classifying sub-sequences of object requests into predefined categories *sequence classification* problem. In this Chapter, our DeepCache framework focuses on predicting *object popularities* as it is one of the most important object characteristics by predicting the probabilities of future object requests for the different timescales mentioned above.

4.3.2 Content Popularity Prediction Model

Recurrent neural networks (RNN) have shown their dominance to handle the problem of seq2seq modeling. In our DeepCache framework, we focus on long short term memory (LSTM) [41] networks which is a special kind of RNN that is capable of capturing long-term and short-term dependencies. LSTM models are widely used due to their special design which allows them to avoid vanishing and exploding gradient problem while building deep layer neural network models. We use LSTM Encoder-Decoder model as shown in Fig. 4.2 for seq2seq prediction, the encoder part encodes the input sequence into a hidden state vector from which the output sequence is then decoded by the decoder part. The main challenge of our DeepCache is the design on the appropriate input features for the ML model to predict useful output sequence to use for caching decisions using LSTM.

Input: Let $X_t = \{x_1, x_2, ..., x_t\}$ be a sequence of objects requested so far at time t where each $x_t \in \mathbb{R}^d$ represents the input feature vector ($d-$dimensional) corresponding

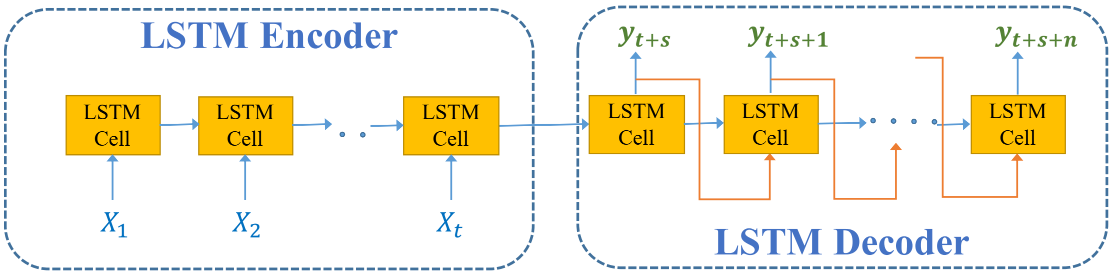


Figure 4.2: LSTM Encoder-Decoder Model used in DeepCache framework. We have an input sequence of request objects $\{x_1, x_2, ..., x_t\}$ at time t and desired output sequence $\{y_{t+s}, y_{t+s+1}, ..., y_{t+s+n}\}$ where $s > 0$ represent the shift in time and $n > 0$ is the desired number of outputs.

to an object. We construct x_t as the probability vector of all unique objects at time t computed in a predefined probability window. The definition of probability window can either be time-based or have a fixed-length size. The input dimension d is equal to the number of unique objects.

Output: Let $Y_t = \{y_1, y_2, ..., y_n\}$ be the sequence of n outputs associated with the arrival of object x_t where $y_n \in \mathbb{R}^p$ represents the output feature vector of $p-$dimension. Our output Y_t at time t, is a sequence of n future probabilities, where n represents the number of probabilities to predict – possibly at multiple time steps of nearby and long-term future probabilities. Here again, output dimension p would be equal to the number of unique objects *i.e.*, $p = d$.

We primarily focus on constructing Y_t to be future object popularities based on past X_t popularities. We construct the input X_t from a given trace of object requests (*i.e.*, a workload from our datasets §4.4.1), and split the data further into training and testing parts. We train our LSTM Encoder-Decoder model to predict future probabilities of any requested object at time t. For better performance of LSTM, we provide multiple past probabilities (denoted as m number of past probabilities) as input. Each of these probabilities are calculated using a predefined probability window. As a result, our input and output can be seen as a 3D Tensor with dimension $(\#samples, m, d)$ and $(\#samples, n, d)$, respectively. We found that LSTM encoder-decoder performs much better if we separately feed the probabilities of each object as a sample data (instead of appending them in an input feature vector). This results in an input and output tensor with dimension $(\#samples * d, m, 1)$ and $(\#samples * d, n, 1)$, respectively. The reason is that in our datasets, time series of object popularities are independent of each other. However, we expect the former mentioned input/output data construction to work better in case the popularity of objects are correlated over time.

4.3.3 Caching Policy

The *Caching Policy* component gets the characteristics predicted by the Object Characteristics Predictor (see Fig. 4.1), and makes decisions on what to cache or evict. In this work, we consider content popularity as the object characteristic that is being predicted. Therefore, Content Popularity Prediction Model predicts future object popularity. We design a *simple* yet *smart* caching policy in a way to make DeepCache interoperate with traditional caching strategies such as LRU, LFU, ... etc The main *novel* idea of this caching policy is that given the future content popularities from the predictor component, our caching policy generates "*fake content requests*" and forwards them to the integral operator. The integral operator is a simple *merge* operator, which merges a stream of fake request and the original request, then sends them to the cache. Such fake requests would make the traditional cache (*e.g.*, LRU) prefetch these objects.

With accurate content popularity prediction, the fake requests indeed represent the future popular objects, and hence prefetching them before their real requests leads to increasing the overall number of cache hits. Depending upon the definition of probability window, we can either generate such "fake content requests" periodically over time or for every i^{th} request. Our evaluation considers both approaches. Using fake requests would lead to additional cache evictions and additions – thus increasing overall network and system load. However, this is the trade-off to increase the cache efficiency, and minimize the user latency. Finally, Fig. 4.3 shows how we piece together all the different components in our DeepCache framework to improve cache efficiency.

4.4 Evaluation

We evaluate our DeepCache framework using two synthetic datasets with different characteristics. In this section, we explain the data generation process of the synthetic datasets, and show the results of applying DeepCache to them under different settings.

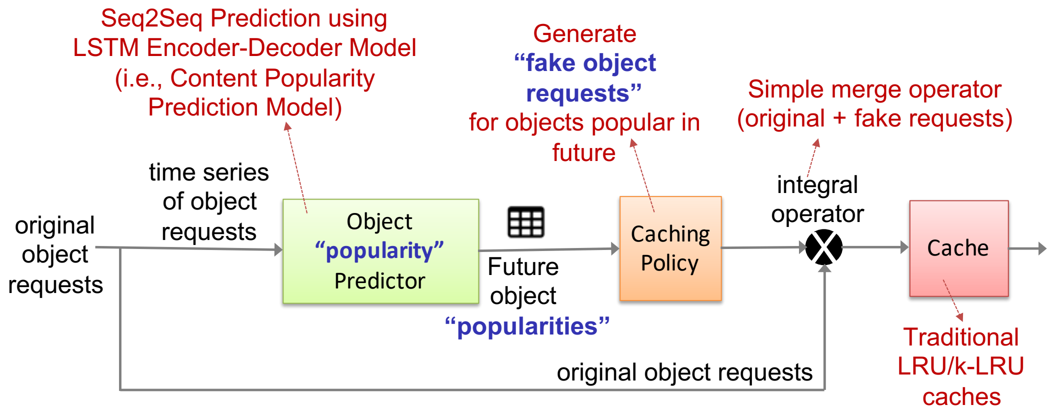

Seq2Seq Prediction using LSTM Encoder-Decoder Model (i.e., Content Popularity Prediction Model)
Generate "fake object requests" for objects popular in future
Simple merge operator (original + fake requests)
time series of object requests
original object requests
Object "popularity" Predictor
Future object "popularities"
Caching Policy
integral operator
Cache
original object requests
Traditional LRU/k-LRU caches

Figure 4.3: A Case For DeepCache

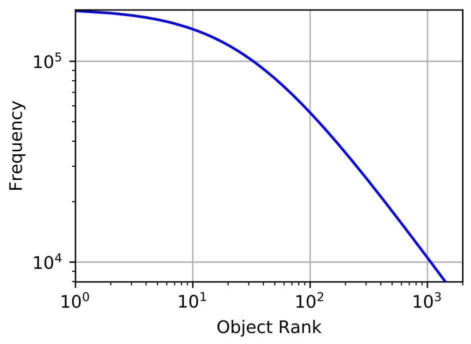
4.4.1 Data Generation

Dataset 1: Dataset 1 has 50 unique objects with more than 80K requests. It includes six intervals of time series which primarily differ in object popularities, and within each interval object popularity ranks remain constant. Object popularities are generated using Zipf distributions with $\alpha = [0.8, 1, 0.5, 0.7, 1.2, 0.6]$ as the parameters for each interval for 50 objects. The popularity rank of objects is generated by a random permutation for each interval.

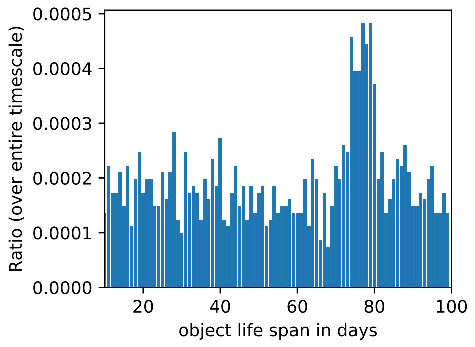
Dataset 2: For a more realistic workload, we use MediSyn [36] to create Dataset 2. This dataset has a total of 1,425 unique objects with more than 2 million requests. The workload has static properties as well as temporal properties denoted using `S:` and `T:` prefixes, respectively. To implement the temporal properties of the workload, we assume that each object has a life span, all the object requests follow a diurnal pattern, and the access ratio to an object diminishes each day.

- `S:`**Object Frequency:** We use a *generalized Zipf distribution* to generate object's frequencies. We use $\alpha = 0.8$ as Zipf distribution exponent parameter, $M = 175,000$ as the maximum frequency, and the scale parameter $k = 30$ for $1,425$ objects. (See Fig. 4.4a for Object Rank vs. Object Frequency).

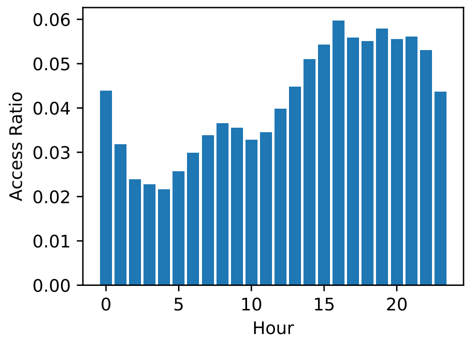
- `T:`**Object Life Span:** We define object life span as the number of days the object



(a) Object Popularity



(b) Object Life Span



(c) Hourly Access Ratio

Figure 4.4: Workload Properties of Dataset 2

Table 4.1: Object Life Span Parameters

Normal dist. Parameters	lognormal μ	lognormal σ
μ	3.0935	1.1417
σ	0.9612	0.3067

is seen during the whole time series of the workload. We use *log-normal distribution* to generate object life spans. For dynamic generation of life spans, the parameters of log-normal distribution, μ(mean) and σ(standard deviation), are generated by two normal distributions. The parameters are stated in Table 4.1. The distribution of the generated life spans for objects is shown in Fig. 4.4b.

- `T`:**Object Access Rate:** We use non-linear functions to control the request access rates for an object during its life span. The number of requests received for an object diminishes each day.

- `T:`**Diurnal Pattern:** The diurnal pattern for each object's request arrival process within a given day is modeled as a *non-homogeneous Poisson process*. Each bin, which is an hour, has a specified Poisson parameter. Fig. 4.4c shows the diurnal hourly ratio values of requests per day. The number of requests for an object for each day of its life span is specified by its Object Access Rate mentioned above.

4.4.2 Experimental Setup

LSTM Encoder-Decoder Model Settings: For our datasets, we use a two-layer depth LSTM Encoder-Decoder model with 128 and 64 as the number of hidden units. All experiments were run on a $2\times$ GPU TITAN V. The loss function is chosen as mean-squared-error (MSE). We ran our experiments for a number of epochs equal to 30, with the batch size set to 10% of the training data. Runtime for all of our experiments is confined within the period of $30 - 120$ minutes.

LSTM Input-Output Data Construction Settings: The sample sequence length is set to be 20, (*i.e.*, $m = 20$ number of past probabilities) for both datasets. For dataset 1, the probability of object o^i is calculated as $a_i/1000$, where a_i represents the number of occurrences of o^i in the window of the past 1K requests of the workload (fixed-length probability window size). While for dataset 2, the probability of o^i is the normalized frequency of that object in an hour (time-based probability window size). In dataset 1, we aim to predict the next $n = 10$ future probabilities (*i.e.*, 10 future time units). For dataset 2, we set $n = 26$, but only used subset of these predicted probabilities. For both datasets 1 and 2, we used 80% of the content objects for training, and the remaining 20% objects for testing. The entire life span of objects from the training set were used to train our model. Testing was conducted on content objects which have never been seen by the model. We then use the predicted future probabilities to make caching decisions.

Cache Policy Settings: As discussed earlier, we use a simple caching policy to enable DeepCache to interoperate with traditional caches. For every object request o_t^i at time

Table 4.2: Prediction Accuracy.

	MSE	**MAE**
Dataset 1	1.2×10^{-6}	4.8×10^{-3}
Dataset 2	3.8×10^{-6}	8.3×10^{-3}

t, we generate a varying number of "fake object requests" (denoted as F_t). For dataset 1, we generate F_t by calculating the top $L = 5$ objects with highest probability at $t+1$. For dataset 2, we rather consider a varying number of top L objects with the highest probability from multiple time intervals. In other words, our fake set of requests F_t not only considers immediate future, but also considers popular objects in the next 12 hours and 24 hours with a diminishing weight for the number of selected objects from each interval.

Integral Operator: For both datasets, the operator is a simple *merge* operator, where the actual object request is followed by all the fake requests generated by our *Caching Policy*. This helps us to update the state of the cache by prefetching objects based on future object popularity and evict unpopular ones.

Cache: For dataset 1, we set the cache size to 5, while for dataset 2 we set the cache size to 150.

4.4.3 Experimental Results

LSTM Encoder-Decoder Prediction Accuracy: Table 4.2 shows the mean-squared-error (MSE) and mean-absolute-error (MAE) of our prediction accuracy on Datasets 1 and 2, both ranging from 0 to 1 in our case. These low error rates show the strong performance of our LSTM model for object popularity prediction. To give a sense of our predictions, Fig. 4.5 shows the ability of LSTM predicting the next i^{th} hourly count for object requests. As evident from Fig. 4.5, LSTM performs quite well in tracking the original time series over multiple future time steps.

Cache-Hit Efficiency: Fig. 4.6 shows the result of applying a simple form of DeepCache Framework on both Datasets 1 and 2. For instance, in Fig. 4.6a, we compare traditional LRU with DeepCache, and without DeepCache. P-Optimal shows the

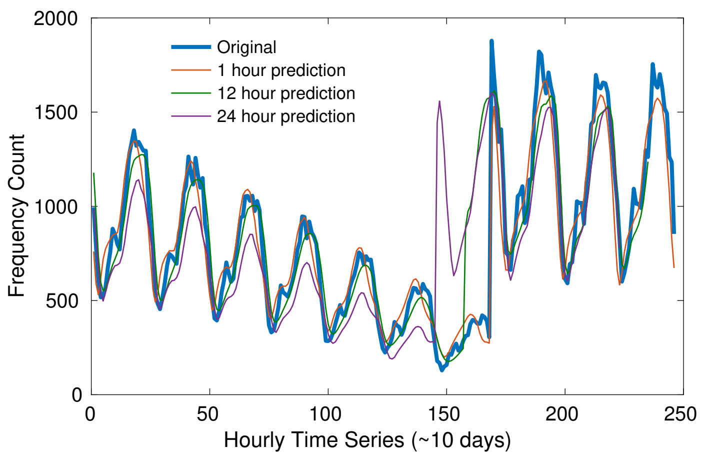


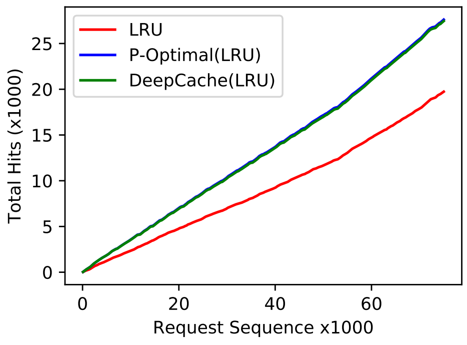
Figure 4.5: Performance of our LSTM-based Content Popularity Prediction Model of an object. Here, we see LSTM performs well for predicting $i^{th} = \{1, 12, 24\}$ hour ahead of time in comparison with the original values over a time series of ∼10 days.

performance of DeepCache with 100% accuracy in content popularity prediction. For Dataset 2 which represents a more realistic workload with large number of object catalog and cache size, we evaluate DeepCache using both LRU (see Fig. 4.6b) and K-LRU (see Fig. 4.6c). In K-LRU, the object has to traverse $K - 1$ virtual caches before it is inserted in the physical cache [19], we used $K = 3$ in our experiments. For all experiments, we found DeepCache significantly outperforms the traditional caching policies LRU, K-LRU. Surprisingly, in Fig. 4.6c, we observe that DeepCache with K-LRU has slightly higher cache-hit than P-Optimal. We hypothesize this is due to LSTM's smooth probability prediction behavior. In case of P-Optimal, probabilities are frequently changing, which makes caching policy less stable compared to DeepCache. As a result, slightly more cache hits are observed for DeepCache over longer period of time.

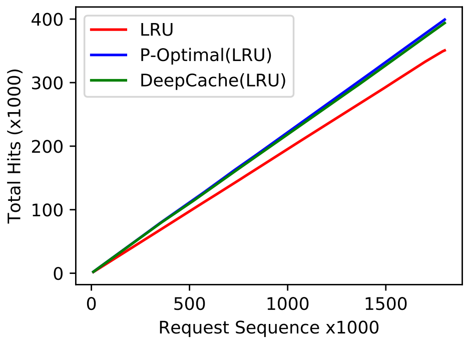
4.5 Discussion

Here, we provide additional insights about expanding DeepCache on two other dimensions: i) learning inter-object dependencies, and ii) account for a hierarchy/network of cache servers.

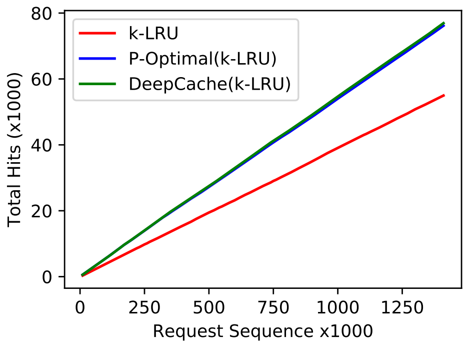
Hierarchical Cache Network. DeepCache in its present form works locally on a



(a) LRU on Dataset 1



(b) LRU on Dataset 2



(c) K-LRU on Dataset 2

Figure 4.6: Cache Hit Performance using DeepCache

single cache. Applying it to a network of caches introduces some issues. One of the main challenges in a hierarchical cache network is the difficulty to estimate user request patterns at intermediate caches: considering a line of caches as a simple case where each cache applies its caching policy independently from the other caches, we can notice that even if the object request streams at the edge cache are independent, the request arrival streams at higher layer caches are no longer independent, because they are generated by the cache misses from the lower layer caches. This dependency between caches makes the estimation of user request patterns at the higher layers a challenging task.

Our work in Chapter 3 addresses this problem by introducing the “BIG” cache abstraction, which views a line of (allotted portions of) caches along the path from an edge server passing by some intermediate servers till the origin server as one virtual “BIG” cache. Thus, any caching policy can be applied to this virtual "BIG" cache as a *single consistent strategy* improving their performance and cache utilization as illustrated. However, applying existing caching policies to "BIG" cache still relies on the received requests at the edge server to decide which objects to cache and evict, regardless of the future object popularity. This leads to caching objects which might be evicted before their next request due to changes in their popularity as discussed before. This is where DeepCache helps fill the gap by combining DeepCache's content popularity prediction model with "BIG" cache abstraction to better decide which objects to cache and evict in a network of caches.

Object Dependency Pattern Prediction Model. The problem of learning object-dependency is more challenging as objects are assumed to be governed by independent inter-arrival processes without any regard to their order. But in real world, there exists strong dependency patterns among objects which are related (*e.g.*, objects in the same webpage). For this purpose, this problem can be formulated as predicting the next sub-sequence of objects based on the past seen (sub-sequence of) objects instead of directly learning the object-dependency. This formulation implicitly accounts for object-dependency and directly gives the predicted objects that can be utilized by the caching policy.

As huge amount of cache access corpus is already available for a server, thus standard *word2vec* [45] model can be deployed to learn feature representation of content objects. As a result, embedding layer can be considered as a first layer of the deep learning model. In the next layer, a bi-directional LSTM [46] (Bi-LSTM) can be adopted as the encoder to capture the sequential relation between content objects. In general, Bi-LSTM model tends to perform better than uni-direction LSTM for capturing interaction among long sequences. In recent years, attention mechanism [47] has been proven to be essential layer in order to further boost the representation power of sequence learning. Basically, attention mechanism helps to identify those input tokens (*i.e.*, content objects in our case) which are more important in sequence prediction by associating an attention weight to each token. Hence, a self-attention layer can be added on top of Bi-LSTM layer encoder which can subsequently be fed to another Bi-LSTM decoder model. The whole process encodes input sequence into a hidden state vector using attention mechanism, and the decoder part decodes the hidden state vector into predicting the next sequence of objects.

4.6 Summary

We proposed DeepCache Framework, a paradigm which uses the state-of-the-art machine learning tools to improve the performance of content caching. Using such a framework, we proposed how to formulate the object characteristics prediction problem as a seq2seq modeling problem. We successfully showed the ability of our LSTM based models to predict the popularity of content objects. To show the efficacy of our approach, we evaluated it using two synthetic datasets under multiple settings out of which one tries to emulate realistic workloads. Our results show that enabling DeepCache with existing cache replacement algorithms such as LRU, K-LRU significantly improves their performance. We also discussed how DeepCache framework can be combined with "BIG" cache to be applied for a hierarchical network of cache servers.

Chapter 5

Resilient Routing

5.1 Introduction

As today's networks grow in scale and complexity, the probability of network failures has increased significantly. It is reported in [48] that multiple failures occur on a daily basis in large data center networks, whereas it has been long known that link failures occur frequently in carrier networks [49, 50, 51]. The latter studies have demonstrated that the conventional IP routing protocols (*e.g.*, using IS-IS/OSPF) that react to failures via *reroute recomputations* can take 10ms or 100 ms to converge. The slow convergence coupled with transient loops caused by *inconsistent* routing updates at different routers may lead to millions of packet losses [49, 51, 52, 53]. This has led to the development of various IP Fast Rerouting, schemes (see, *e.g.*, [54, 55, 56, 57, 53, 58, 59, 60] and references therein). These schemes as well as the traditional 1+1 path/link protection schemes used in optical/(G)MPLS networks are designed primarily for protecting against a single link failure; apart from the work in [54, 55, 56], most are heuristics that do not work in all *single* link failure scenarios (see §5.2 for an overview of the state-of-the-art).

The rise of 5G, edge computing, Internet-of-Things (IoTs) and cyber-physical systems will further expand the scale, geographical span and complexity of networked systems, while ensuring high reliability and resilience of these systems become ever more critical. The growing scale, span and complexity, together with the adoption of network function virtualization, will likely not only lead to further increases in network failures (*e.g.*, due to software bugs and crashes), but also produce more complex failure scenarios that go beyond single link/node failures. *Resilient routing against arbitrary network failures* – namely, the ability to *proactively prepare for any number of link/node failures* (not merely a single link/node failure) via provisioning (*i.e.*, precomputing/installing) certain "forwarding rules" that can guide decision-making in the event of network failures to *dynamically reroute traffic around failures – is therefore imperative.*

Software defined networking (SDN) has made it easier to introduce more sophisticated routing algorithms and forwarding behavior into networks. In a similar vein, so does IETF Segment Routing (SR)[1]. Neither SDN nor SR by themselves provide a resilient routing solution against arbitrary number of network (link/node) failures. In fact, so far only *negative* results are known – no routing algorithms with pre-installed *static* (or *fixed*) forwarding rules are resilient against arbitrary k failures for k as small as two. The only solutions are either designed for *specific* topologies (*e.g.*, [61, 62, 63, 64, 65, 66, 67]), or resort to *blindly* search for and *dynamically* reconstruct a new feasible route *after* failures [68, 69, 70, 71] – they are essentially *reactive* and may take long time to converge (see §5.2 for more discussion).

In this chapter, we develop the first *provably correct proactive routing algorithm that is resilient against arbitrary network failures*: Given any arbitrary number of link/node failures in a network (with any general network topology), as long as the link/node failures do not disconnect a source node s from a destination node d, our algorithm is guaranteed to route around the failures and successfully forward a packet from s to d. Unlike most existing routing algorithms which rely on computing shortest path (or "breadth first search") spanning trees for routing, our algorithm is based on *depth first search* (DFS) spanning trees (each rooted at a destination node) by intelligently exploiting several of its crucial properties for routing around failures (see §5.3.2). Given any DFS spanning tree of a graph, the "cross-edges" (or the "off-tree" edges) between subtrees connect a node with its ancestor (or descendant) node only. We pre-compute and assign a *monotonic* sequence of (destination-specific) node identifiers (*id*'s), starting with the root/destination *id* set to 1. These node *id*'s encode network connectivity

[1]SR can be viewed as a generalization of the combined IP/MPLS routing paradigm using "source routing" where a packet can encode network topology (and service) path information and instructions in "segments" carried in packet headers. By abandoning the classical destination-based, hop-by-hop, shortest path routing of IP, SR makes it easier to implement proactive resilient routing schemes.

information (especially the *partial ordering* relations induced by the DFS tree); together with appropriately (pre-)installed forwarding rules, they assist nodes in discovering/-maintaining reachability and making *loop-free* forwarding decisions under failures.

The key challenges in utilizing the crucial properties of DFS spanning trees to design a resilient routing algorithm is to judiciously devise a *localized* message exchange and reachability discovery process that avoids *loopy* propagation of *erroneous* or *stale* information; for this, we develop a multi-stage process for *localized, conditioned* message exchanges that utilizes the (original) DFS spanning tree structure and (partially ordered or monotonic) node *id*'s to ensure certain *invariants* are met (see §5.5.2). The message exchange process is *localized* in that message exchanges are limited to nodes (in the DFS subtree(s)) that are affected by the network failures. We have implemented our resilient routing algorithm using NS-3 and conducted experiments to demonstrate the efficacy of our solution and its superiority over other routing schemes under various failure scenarios (see §5.7). As an added advantage our solution decouples (and emphasizes) ensuring *connectivity* (or *reachability*) from (routing) performance, *e.g.*, using a shortest path, and thereby makes it possible to quickly find alternative "next-hops" for connectivity/reachability under arbitrary failures. It is possible to augment our solution with additional (routing) metrics (*e.g.*, in terms of bandwidth, latency or other QoS metrics) to combine reachability and routing performance (see §5.8 for a brief discussion).

5.2 Towards a Theory of Resilient Routing: State-of-the-Art

5.2.1 From Reactive Routing to Proactive, Resilient (Fast Re-)Routing

The conventional IP routing protocols are based on *shortest paths* and fall broadly into two categories *distance vector* (DV) or *link state* (LV). In terms of dealing with network (node/link) failures, both are *reactive* in that they resort to (shortest path) *route recomputation after detecting or informed of network failures.* In particular, the DV routing protocols (*e.g.*, RIP) that employ the Bellman-Ford algorithm for (*distributed*) short path computations suffers from the well-known "count-to-infinity" problem. Besides certain "hacks" (*e.g.*, Poisonous Reverse implemented in RIP) that do not work in all topological settings, the only known solution that circumvents the "count-to-infinity" problem is the Diffusing Update algorithm (DUAL) [72] that employs a complicated (*recursive*) "query-reply" process[2] known as *diffusing computation* for route (or "nexthop") recomputations to resolve potential routing loops.

In contrast, LS-based routing protocols (*e.g.*, OSPF and IS-IS) employ link state flooding to first build a topology database and then compute a shortest path to each destination (*e.g.*, using Dijkstra's algorithm) independently of other nodes. When a network (link/node) failure is detected, a node sends a link-state update to all other nodes, which triggers a short path recomputation at all nodes. While this avoids the count-to-infinity problem suffered by DV algorithms, *transient* loops may still occurs[3]. These, coupled with the *slow* convergence of LS route recomputation (which hinges critically on the link state updating and propagation times, and may take 10ms or 100 ms in a large network) may result thereof due to *inconsistent* routing state may lead to millions of packet losses over a high-speed link [49, 51, 52, 53].

The slow convergence of conventional IP routing protocols has led to the development of various IP Fast Rerouting schemes with pre-computed "backup" next-hops for packet forwarding under failures without resort to route recomputations [54, 55, 56, 57, 53, 58, 59, 60] and references therein. Most of them are heuristic schemes that do not work in all *single* link failure scenarios. The challenges lie in ensuring *consistency* in *distributed* routing states while also obeying the destination-based, hop-by-hop IP shortest path routing paradigm. In contrast, proactively provisioning (*i.e.*, setting-up) one or more backup paths can be readily done in optical or MPLS networks that employ circuit switching or virtual circuits to set up one or more paths between any source-destination pair, with notions such as 1+1 path or link protections. Nonetheless, the 1+1 path/link protection approaches employed in optical/MPLS networks and IP Fast Rerouting schemes are designed primarily to protect a *single* link (and sometimes a *single* node) failures, and in general cannot handle multiple network failures.

[2]As pointed out in [68], DUAL cannot handle multiple (concurrent or successive) failures at the same time. The authors in [68] establish new *invariant* properties for successor computation and develop a simplified procedure that allows multiple asynchronous diffusing computations.

[3]Unless orderly route updates [73, 74] are performed, transient forwarding loops can occur in a distributed [49, 55] & centralized routing setting [75, 74].

The emergence of software-defined networking (SDN) has renewed interests in proactive, resilient routing for a number of reasons. With a logically centralized control plane and a more flexible programmable data plane, SDN makes it easier to employ more sophisticated (centralized) routing algorithms and control switch forwarding behavior by (pre-)installing *(match-action) flow rules*. This is achieved without the need to upgrade and deploy "new" protocols in routers/switches; just as in an MPLS network, any path (not only shortest paths) can be used for routing and forwarding, provided that the installed flow rules at the switches are consistent, *i.e.*, no loops. On the other hand, the centralized control plane of SDN also accentuates the need for resilient routing to protect, *e.g.*, the "control paths" between a SDN controller and the switches in the data plane or those among multiple controller instances to ensure correct, consistent and timely SDN control operations. As eloquently argued in [70], moving the responsibility of ensuring connectivity to the data plane by endowing switches with *local rerouting* capabilities is critical to SDN. Its importance is further illustrated in [76] where it is shown that the control path failures among multiple SDN controller instances may lead to the failure of Raft, a consensus protocol used in ONOS to ensure strong control state consistency.

To ensure data plane connectivity, the authors in [70] develop the DDC scheme based on the classical *link reversal* algorithm [69] which dynamically (re-)reconstructs a *directed acyclic graph* (DAG) per destination under failures using so-called link reversal operations. Basically, when encountering a link failure, each node dynamically "searches" for a new DAG to a destination by reversing the direction(s) of one or all of its links. If there is a path between the node and the destination, the algorithm is guaranteed to converge, but in the worst case may incur $O(n^2)$ link reversal operations [77]. If the node has lost connectivity to the destination, the link reversal process will not stop [77], suffering a "count-to-infinity"-like problem. The authors in [71] cleverly implement a dynamic graph search algorithm using SDN flow rules and a *dynamic state* carried in packets to iteratively search for an available path upon failures. Other alternative schemes have also been proposed, *e.g.*, via carrying a "blacklist" of encountered failures in packets [78, 79] where a new path is computed *on the fly* by removing the failed links.

5.2.2 Negative Results

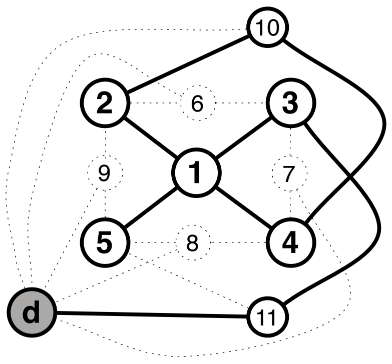
The *failure insensitive routing* (FIR) developed in [56, 54, 55] based on *interface-specific forwarding* (ISF) is the first *provably correct* IP Fast Rerouting algorithm that is guaranteed to be *resilient against any arbitrary single link/node failure*: at each node FIR pre-computes an alternative next-hop per interface – hence its forwarding decision is *interface-specific*; upon detecting a link (or node) failure downstream along the (shortest) path using the (primary) next-hop, it simply reroutes a packet using the (precomputed) alternative next-hop. This result has motivated the authors in [1] to formalize the notion of k*-resiliency* under the assumptions that interface-specific forwarding is employed, and at every node, a set of (interface-specific) *candidate* nexthops may be pre-computed for each destination.

K-Resiliency. Given a target destination d, a (proactive) routing algorithm $\mathcal{R}$ (employed at each node in the network) is said to be k*-resilient* if under any *arbitrary* k link failures that do not disconnect a node s to the destination d, $\mathcal{R}$ is guaranteed to forward a packet from s to d using one of the precomputed candidate nexthops. (Note that given a specific set of k link failures, it is possible that using a precomputed candidate nexthop at a node, say, s, using $\mathcal{R}$ the packet may return back to s again later – in that case a different nexthop may be used by s to forward the packet. As long as the packet eventually reaches its destination d (using one of the pre-computed nexthops), routing is considered successful and thus resilient against the k link failures.)

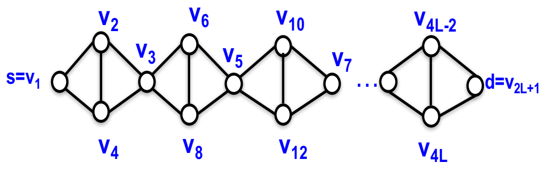
The set of candidate nexthops associated with an interface is said to be *(statically) ordered*, say, denoted by $\langle h_1, ..., h_m \rangle$, if we impose a further constraint on how the set is invoked for packet forwarding[4]: Upon detecting a network failure, when a packet destined to d arrives at the interface for the first time, the nexthop h_1 will be used to forward the packet towards its destination d; when the same packet arrives at the interface for the second time, the nexthop h_2 will be used; this process continues until the nexthop h_m has been used. If the packet returns on the interface again, routing is then deemed to have failed.

Negative Result 1. Using the topology shown in Fig. 5.1a, the authors in [1]

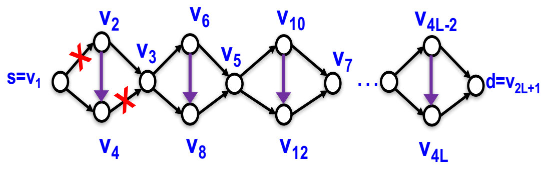
[4]We comment that *interface-specific forwarding* (ISF) is a built-in feature of SDN switches. The *statically ordered constraint* is also an inherent feature of an SDN switch: multiple forwarding (flow) rules installed at each interface that match a packet, say, with destination address d are assigned with priorities, imposing a strict ordering.



(a) Topology 1



(b) Topology 2



(c) DAG Example for Topology 2

Figure 5.1: (a) Example Topology 1 from [1]; (b) Example Topology 2 (c) No pre-configured DAG is resilient against arbitrary two link failures: given the DAG illustrated, there exist failure scenarios (*e.g.*, failures of both (v_1, v_2) and (v_4, v_3)) which render d not reachable from s using the DAG. There are $O(2^L)$ choices in orienting the L vertical links, therefore $O(2^L)$ DAGs.

show that: *No routing algorithm using a pre-computed set of interface-specific, statically ordered candidate nexthops at every node is k-resilient, namely, can proactively protect against arbitrary k link failures for any $k \geq 2$.*

Negative Result 2. Using the topology shown in Fig. 5.1b (with $(2L+1)$-hop shortest paths), we establish another negative result on the number of pre-computed/provisioned paths (or rules) needed to protect against two arbitrary link failures[5]: *No routing algorithm which provisions p fixed paths (or more generally p fixed DAGs) with $p = O(L^c)$ is 2-resilient.* The topology shown in Fig. 5.1b can be easily generalized to establish a negative result for k-resiliency with $k > 2$.

[5]In a sense, this negative result is more general than the first one: we implicitly allow forwarding rules at a node to depend on the upstream nodes where a packet has traversed, *e.g.*, marked with a path or DAG id. This negative result suggests that in order to protect against arbitrary two link failures, we may have to pre-install an exponential $O(2^L)$ number of (*fixed*) forwarding (flow) rules in the network.

The above negative results demonstrate that proactively installing a set of (polynomial-sized) *static* or *fixed* forwarding rules are insufficient to protect against arbitrary two or more link failures. On the other extreme, (*dynamic*) routing algorithms such as DUAL, Link Reversal (and its DDC implementation) and Graph Search [68, 69, 70, 71] resort to a generic "exploration" process, whether using diffusing computations, link reversals or graph search – to blindly search for a feasible solution when encountering failures, incurring long convergence times in the worst case. In particular, these algorithms do not make avail of any topological structure information (that may be "encoded" via certain pre-computed/installed rules) to help guide the exploration process. This gives rise to the following important question:

Is it possible to develop a k-resilient routing algorithm by pre-computing a set of rules (and other relevant information) for packet forwarding, and upon network failures, dynamically modifying/adjusting the rules (e.g., by filtering and re-ordering the rules) based on detected or inferred failure scenarios? More generally and ambitiously, *is it possible to design a resilient routing algorithm against arbitrary network failures?* Namely, it is capable of successfully routing a packet from its source s to its destination d as long as a path from s to d exists after an arbitrary number of link/node failures.

5.3 Depth-First Search Trees and Connectivity Encoding

Conventional routing algorithms are designed primarily using shortest path spanning trees – a generalization of *breadth-first-search* (BFS) trees for weighted graphs – with an emphasis on routing performance. In the following we illustrate why *depth-first-search* (DFS) spanning trees would be a better alternative for maintaining *connectivity* or *reachability* under network failures.

5.3.1 BFS vs DFS: An Illustration

We will use the network topology in Fig. 5.2 to provide an intuitive illustration of the key differences in BFS vs. DFS spanning trees, especially under failures. We will then provide formal notations and statements regarding the critical properties of DFS (spanning) trees of a general graph.

Fig. 5.2b shows a BFS spanning tree of the graph in Fig. 5.2a rooted at node A

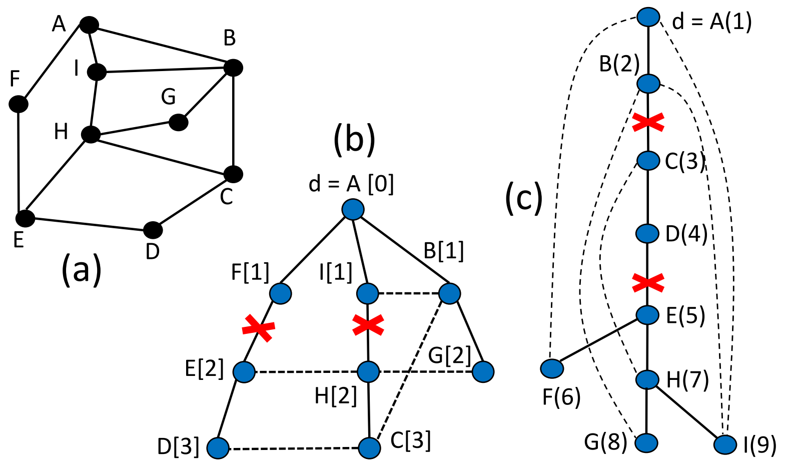


Figure 5.2: For the simple network in (a), routing path recovery using BFS tree in (b), and routing path recovery using DFS tree in (c) (best viewed in color).

(the destination), whereas Fig. 5.2c shows a DFS spanning tree for the same graph rooted also at node A. In both figures, the dark directed arrows (lines) indicate the "on-tree" edges (from a child to its parent) and the dashed arcs represent an "off-tree" edges or *cross* edges/links (between non-parent-child nodes). The number in the *square bracket* besides a node in Fig. 5.2b indicates the distance (in this case, hops) to the root/destination A: all nodes with the same number lie at the same level of the BFS tree. In contrast, the number in the *parenthesis* besides a node in Fig. 5.2c denotes the node identifier (id) assigned to it based on the steps it takes the DFS to visit the node starting from the root A (with id=1). Its significance will be expounded on below. One key difference between BFS and DFS spanning trees is obvious when we examine the *cross* edges: (i) in a BFS tree, cross edges only occur between nodes (either at the same level or between different levels) on two *different* tree branches from the root; whereas (ii) in a DFS tree, cross edges only occur between nodes on the *same* tree branch from the root, connecting a node with either one of its ancestor node (with a smaller node id) or a descendant node (with a larger id).

First consider the BFS tree in Fig. 5.2b. Suppose two "on-tree" edges, say, $F - E$ and $I - H$ have failed, neither E or H can reach A via its parent (the default next-hop) to reach A, but both are connected via the cross edge $E - H$. The BFS tree structure does not provide an obvious way for H to know that it can still reach A via the cross edge G but not via the cross edge $E - H$. For example, upon receiving a packet from C to A, if H decides to forward to E via the cross edge $E - H$, a forwarding loop may occur. Node H has to resort to a "trial-&-error" search to find an "escape path" to A. If in addition, the "on-tree" edge $A - B$ and "off-tree" edge $I - B$ have failed, nodes E and H do not have an easy way to determine that they have lost connectivity to A with exploring all possible paths. Now consider the DFS tree in Fig. 5.2c. Suppose two "on-tree" edges, say, $B - C$ and $D - E$, have failed. Node B can learn from one of its (connected) descendants, say, via I, that it is still connected to A. Likewise, node E can learn from its descendants, *e.g.*, F, that it is connected to A. Suppose in addition two "off-tree" edges $A - F$ and $A - I$ have also failed, each node (and its descendants) will learn that it has lost connectivity to A as none of its descendants are connected to a node whose id is smaller than $id(B)$.

5.3.2 Critical Properties of DFS Trees

We now formally state several critical properties of DFS trees that we exploit for the design of our resilient routing algorithm under arbitrary failures. We denote a network topology by a graph $G = (V, E)$, where V is the set of vertices/nodes $|V| = n$ and E is the set of edges/links. For any node $v \in V$, we use $N(v)$ to denote its neighbors. For the routing purpose, a DFS spanning tree is defined per destination. Hereafter for simplicity of exposition, we will fix a destination, denoted by d, and consider an (arbitrarily generated) DFS spanning tree $T^{(d)}$ (or simply T) rooted at d. We denote the *tree* edges by $E^T \subset E$ (with $|E^T| = n-1$), and refer to all edges in $E \backslash E^T$ as "off-tree" edges or cross edges/links. We will use vertices/nodes and edges/links interchangeably. Although any tree induces a partial ordering on the node set V, the *partial ordering* $\prec_T$ induced by a DFS spanning tree T has several critical properties that are not enjoyed by other types of spanning trees such as BFS trees. Similar statements proving the following properties can be found in [80] for example.

Property 1. Properties of the Monotonic Node ID Sequence of a DFS Tree. For the sake of notional simplicity (and practical implementation), instead of using the *partial ordering* $\prec_T$, we assign a sequence of *monotonically increasing* node identifiers (id's) to the nodes, starting with the root d with $id(d) = 1$. For $u \neq d \in V$, its node id is assigned based on the order it is visited during the process of generating the DFS spanning tree T. Clearly, if u is the parent of v in T, $id(u) < id(v)$. We will use $ANC(v)$ to denote the set of ancestors (including the parent) of node v, *i.e.*, all nodes reside on the downstream branch (path) from v to the root d in T. Clearly, $id(v) > id(a)$ for any $a \in ANC(v)$. We will use $DES(v)$ to denote the set of descendants of node v, *i.e.*, all nodes reside in a subtree rooted at v. Clearly, $id(v) < id(x)$ for any $x \in DES(v)$. Furthermore, if v and w are both children of u, but $id(v) < id(w)$ (*i.e.*, v is visited before w), then for any $x \in DES(v)$, $id(x) < id(w)$, and for any $y \in DES(w)$, $id(x) < id(y)$.

Property 2. Properties of Cross Edges in a DFS Tree. For any *cross* edge $e = (x, y) \in E \setminus E^T$ (thus x is neither the parent nor a child of y), we have either (i) $x \in ANC(y)$ and thus $id(x) < id(y)$, or (ii) $x \in DES(y)$ and thus $id(x) > id(y)$. In other words, any *cross* edge connects an ancestor with a descendant lying on the *same* branch – there are no *cross-branch* "off-tree" edges in a DFS tree!

Corollary 1. Let w_1 and w_2 share a common ancestor (*i.e.*, $w_1 \notin ANC(w_2)$ or $w_2 \notin ANC(w_1)$). Then, for any $x \in DES(w_1)$ and $y \in DES(w_2)$ $(x, y) \notin E$. Namely, there is no cross edge connecting any two sibling sub-trees.

In summary, these properties essentially state that the *connectivity* structure of a network topology is encoded by the DFS tree and the monotonic node id sequence (partial ordering) it induces: Given any node s, it can either reach the root (destination) d via a path going through its parent node in the DFS tree, or via one of its children or descendants which has a cross edge to one of its ancestors. Hence when network failures disrupt its connectivity to d via its parent, the key to route around failures is then to *discover the existence of a descendant node which is connected to an ancestor that is not affected by the failures.* However, *this discovery process is non-trial* – we present the major challenges and highlight the key principles and ideas in tackling them next section.

5.4 Handling Arbitrary Failures: Challenges & Solutions

We first introduce further terminologies to characterize a network and its DFS spanning tree under an arbitrary number of link failures. We then illustrate the major challenges in discovering and maintaining reachability after failures, and discuss key ideas in circumventing them.

5.4.1 Islands, Bridges, and Gateways

Throughout the chapter, we will consider only link/edge failures, as a node failure can be treated equivalently as all its adjacent edges have failed. Given a network $G = (V, E)$ and a DFS spanning tree with the tree edge set E^T, we refer to the failure of an edge in E^T as an *on-tree* link failure, and the failure of an edge in $E \setminus E^T$ as an *off-tree* link failure. Note that if all link failures are off-tree, then *reachability* to root/destination d as encoded by the DFS tree T is still intact. Hence given a set of link failures $F \subset E$, we assume at least one "on-tree" edge has failed, *i.e.*, $F \cap E^T \neq \emptyset$. Hence the failures F will break T into a forest consisting of multiple (now *disconnected*) sub-trees of T, denoted by $T_0, T_1, \ldots, T_K$, $K = |F \cap E^T| \geq 1$. In the following, unless explicitly stated, when an edge e (either "on-tree" or "off-tree") is mentioned, it is assumed to be a *non-failing* edge, *i.e.*, $e \in E \setminus F$. Clearly, for each T_i, the failures F only affect the "cross" edges between nodes within T_i (otherwise T_i is broken into two disconnected subtrees); in other words, $F \cap E^{T_i} = \emptyset$ for $0 \leq i \leq K$.

We will always use T_0 to represent the subtree containing d (thus rooted at $r_0 = d$), and refer to it (together with any (non-failing) "off-tree" edges among the nodes in T_0) as the *mainland* (or main component, component 0). We will refer to other subtrees with "off-tree" edges among the nodes within each subtree T_i, $i \geq 1$, as an *island*. Hence for each island T_i, its (local) root r_i has lost its connectivity to its parent in T, *i.e.*, the edge connecting r_i to its parent has failed. For any two subtrees, T_i and T_j, $0 \leq i \neq j \leq K$, if there exists an "off-tree" or cross edge $e = (x, y)$ connecting T_i and T_j, *i.e.*, $x \in T_i$ and $y \in T_j$, we refer to e as a *bridge* (between T_i and T_j). The nodes x and y are called *gateways* (to the other subtrees). We will refer to a collection of islands that are connected via bridges as an *archipelago*.

Property 3. Constraints on Bridged Islands and Properties of Bridges and Gateways. Property 2 and Corollary 1 place constraints on two *bridged islands*, T_i and T_j. Assume $id(r_i) < id(r_j)$. If there exists a a bridge $e = (x, y) \in E \setminus F$, $x \in T_i$ and $y \in T_j$, connecting T_i and T_j, then r_i must be an ancestor of r_j in T, *i.e.*, $r_i \in ANC(r_j)$ (or $r_i \prec_T r_j$), and $id(r_i) \leq id(x) < id(r_j) \leq id(y)$.

Clearly, an island T_i is still connected to the mainland T_0 after the failures F if and only if i) there exists a bridge between T_i and T_0; or ii) T_i is part of an archipelago that is connected to the mainland. Hence in order to maintain reachability from nodes in T_i

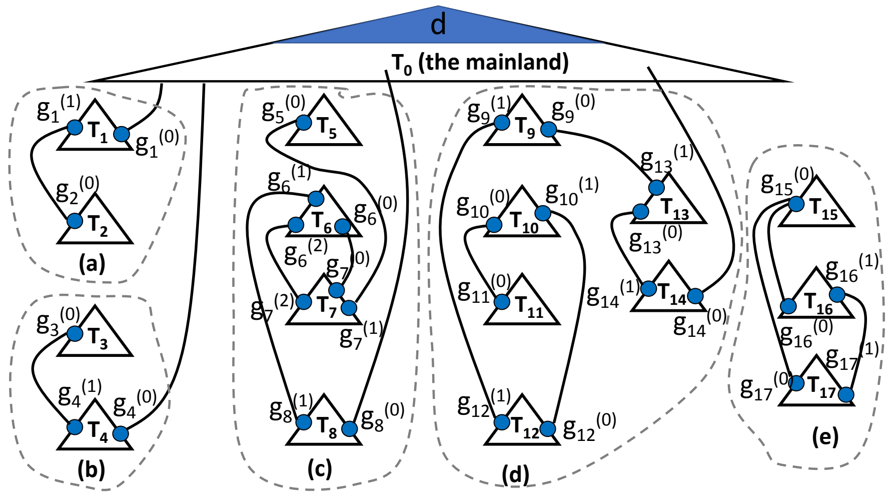

d
T_0 (the mainland)
$g_1^{(1)}$ T_1 $g_1^{(0)}$
$g_2^{(0)}$ T_2
(a)
$g_3^{(0)}$ T_3
$g_4^{(1)}$ T_4 $g_4^{(0)}$
(b)
$g_5^{(0)}$ T_5
$g_6^{(1)}$ T_6 $g_6^{(0)}$ $g_6^{(2)}$
$g_7^{(0)}$ $g_7^{(2)}$ T_7 $g_7^{(1)}$
$g_8^{(1)}$ T_8 $g_8^{(0)}$
(c)
$g_9^{(1)}$ T_9 $g_9^{(0)}$
$g_{10}^{(0)}$ T_{10} $g_{10}^{(1)}$
$g_{11}^{(0)}$ T_{11}
$g_{12}^{(1)}$ T_{12} $g_{12}^{(0)}$
$g_{13}^{(1)}$ T_{13} $g_{13}^{(0)}$
$g_{14}^{(1)}$ T_{14} $g_{14}^{(0)}$
(d)
$g_{15}^{(0)}$ T_{15}
$g_{16}^{(1)}$ T_{16} $g_{16}^{(0)}$
$g_{17}^{(1)}$ $g_{17}^{(0)}$ T_{17}
(e)

Figure 5.3: Connected and Isolated Archipelagos

to d after the failures, it is crucial to discover the existence of bridges and gateways that connect the island chain with the mainland. We will sometimes refer to a path from a node s in an island to the destination d on the mainland via one or multiple bridges as an *escape* route/path for s. We will call an island/archipelago that does not have any bridge to the mainland an *isolated* island/archipelago.

5.4.2 Discovering Reachability after Failures

Given an island T_i, its (local) root r_i knows that it has lost reachability to d (via the default DFS tree path) because the edge to its parent has failed. Hence it is the responsibility of r_i to (1) notify its descendants (or rather nodes within its subtree) that they are now part of an island and can no longer reach d via r_i; and (2) discover whether any nodes within its subtree is connected to an ancestor of r_i, namely, to a node x with $id(x) < id(r_i)$. However, the general situation can be fairly complicated, when T_i is not directly connected to the mainland T_0, but instead connected to the mainland via an island chain. This will entail passing messages among (connected) islands which poses several challenges. If care is not taken, *loopy* message passing may occur that leads to *non-convergence*!

Fig. 5.3 illustrates multiple (representative) example scenarios of how an island may be (indirectly) connected to the mainland, as well as one example with an isolated archipelago. In archipelago (a) with two islands T_1 and T_2, T_2 is connected to the mainland T_0 via an"up" bridge $(g_2^{(0)}, g_1^{(1)})$ to T_1, an island "above" it, *i.e.*, $id(r_1) < id(r_2)$. In archipelago (b) with two islands T_3 and T_4, T_3 is connected to the mainland T_0 via a "down" bridge $(g_3^{(0)}, g_4^{(1)})$ to T_4, an island "below" it, *i.e.*, $id(r_4) > id(r_3)$. In archipelago (c) with four islands T_5, T_6, T_7 and T_8, T_7 is connected to the mainland T_0 first by traversing an up bridge, *e.g.*, $(g_7^{(2)}, g_6^{(2)})$ to T_6, then a down bridge $(g_6^{(1)}, g_8^{(1)})$ to T_8. Archipelago (d) represents a more complex scenario where an escape path for nodes in T_{11} traverses several "up" and "down" bridges and intermediate islands $(T_{10}, T_{12}, T_9, T_{13}$ and $T_{14})$ before finally reaching the mainland T_0. In archipelagos (c) and (d), there exist multiple bridges connecting the islands which may trap packets in a loop if "wrong" bridges are selected. Lastly, archipelago (e) is isolated from the mainland T_0. But how will the nodes on the islands learn that they have lost connectivity to the mainland, thus can no longer reach the destination d?

The above examples illustrate that the message exchanges among nodes to discover reachability cannot be arbitrary, lest a *loopy* process may ensue where a message may circulate in the network (or rather, archipelago) forever; an *orderly* process must be imposed. Furthermore, "bad" (*stale*, or worst, *erroneous*) reachability information can be quickly identified and filtered, while ensuring the "good" (*i.e.*,*unaffected* by the failures) reachability information is allowed to be rapidly propagated across the archipelago. To tackle these challenges, we design a *multi-stage*, *localized* discovery process with *conditioned* message exchanges that directs good reachability information (gateways with a bridge to the mainland) to rapidly propagate across connected islands while quickly filtering out the circulation of "bad" reachability information so as to enable nodes in an (indirectly connected) island to select the correct, eligible gateways for fast, loop-free packet rerouting and forwarding. To this end, we utilize extensively the properties of the DFS tree and the connectivity information encoded in the monotonic (partially ordered) node id's to ensure certain *invariants* and *conditions* are upheld during the message exchange and reachability discovery process.

We first remark that for nodes within the mainland T_0 (rooted at d with $id(d) = 1$), neither the (global) root d nor any node within T_0 will invoke or be engaged in any message exchange and reachability discovery process (initiated by a local root of an island) described below. For each island T_i, (1) its local root r_i first passes a message (with the local root id, $id(r_i)$) *downward* to inform its children and descendants that are part of a subtree/island rooted at r_i (the *downward* stage); (2) if a node x within T_i has an "off-tree" edge $e = (x, y)$, with $id(y) < id(r_i)$, then node x learns that it is a gateway (with an up bridge) to an island above, and will pass a message *upward* to inform r_i (the *upward* stage). In particular, if T_i is *directly connected* to the mainland T_0, then we have the following crucial property which not only ensures the existence of *good* reachability information for T_i, but also provides an *anchor* that "pins down" the message exchange process and guarantees its quick convergence.

Property 4. If an island T_i is directly connected to the mainland T_0, then there exists a bridge $e = (g_1, g_0)$ with $g_1 \in T_i$ and $g_0 \in T_0$ such that $id(g_0) < id(r_i) \leq id(g_1)$ and in particular, g_0, r_i and g_1 lie on the same branch in T. Clearly, the (default) reachability of g_0 to d is unaffected by the failures F, and thus is not (and will not be) engaged in the message exchange and discovery process. Hence r_i (and other nodes in T_i) will learn that it (they) can reach d via g_1. We will refer to g_1 is an *anchor* gateway for T_i (and the archipelago it is part of).

Now the key question becomes: how do we quickly propagate the learned *good* reachability information across other (connected) islands, while filtering out circulation of "bad" reachability information that does not lead to an escape path to d? This is where we introduce a *conditioned* message exchange process *across bridges* connecting the islands. This stage of message changes will always be initiated by a node $x \in T_i$ with a cross edge $e = (x, y)$ to $y \in T_j$, where $id(x) < id(y) - 1$ (*i.e.*, x is a non-parent ancestor of y). Node x and y will exchange their local root id and the *minimal* node id they have learned so far. First, if both x and y have the same local root id (*i.e.*, $x, y \in T_i$), then the cross edge $e = (x, y)$ is internal to the island, *not a bridge.* We will now restrict the case where $e = (x, y)$ is a bridge and thus x and y learn that they are *gateways* connecting T_i and T_j. The minimum of the two the *minimal* node id's exchanged will be accepted and further propagated, and the other rejected. In particular, unless if T_i is directly connected to the mainland while T_j is not, then Property 2 and Corollary 1 ensure that the minimal id learned by T_i is always smaller than that of T_j (an *invariant*). Using such a conditioned message exchange process, for example, island T_7 in archipelago (c) will (eventually) accept the minimal node id learned from gateway $g_6^{(2)}$ (or $g_6^{(0)}$) in T_6, and reject that from gateway $g_5^{(0)}$ in T_5, whereas in return T_5 will (eventually) learn that it can reach d via gateway $g_7^{(1)}$ to T_7. If a local root r_i finds that all the minimal node id's it has learned from the message exchange and discovery process are larger than its id, it quickly realizes that its island as well as the entire archipelago the island is part of are isolated from the mainland – therefore, there is no "count-to-infinity" problem!

A more formal description of the message exchange and reachability discovery process will be presented in §5.5 with forwarding rules for selecting eligible gateways and loop-free packet forwarding. We emphasize that this process is *localized* in that only nodes within an archipelago are engaged in the message exchanges. In particular, this process does not involve any *recomputing* or *reconstructing* a new (DFS) tree or any other tree, nor are any of the *precomputed* node id's (defined by the DFS tree T) and the parent-child/ancestor-descendant relations (*i.e.*, the partial ordering) induced by T are modified. Instead of "blindly searching" for an escape route, our process is *orderly* and *guided* by the node id's and minimal node id's learned. The goal is to assist the nodes affected by the failures to quickly learn about its *reachability status* by exchanging the *minimal* node id's they know of or have learned so far, and judiciously filter and select eligible gateways to form an escape route for loop-free packet forwarding.

5.5 Loop-Free Resilient Routing

In this section, we present the details of our proposed resilient routing protocol focusing on four aspects: 1) state information stored at each node, 2) rules for updating/maintaining this state, 3) forwarding rules that guarantees no loops are formed during normal operations or upon failures, and 4) Invariants maintained by each node to ensure convergence upon failure. The main idea of the proposed protocol is that upon failures, each node uses its state information to decide whether it is still connected to the dst d or not, and which gateway it can use as its alternative forwarding path.

5.5.1 State Information

Upon the construction of the depth-first search tree T, each node v is initialized with the following during the bootstrap phase: 1) its own $id(v)$, 2) the ids of its neighbors: $id(u)\ \forall u \in N(v)$, 3) its parent $p(v)$, and 4) children $c(v)$. Node v can then define its list of ancestor neighbors $N_{anc}(v) := \{u \in N(v) | id(u) < id(v)\}$, and descendant neighbors

$N_{dec}(v) := \{w \in N(v) | id(w) > id(v)\}$. Each node v also needs to store the $id(.)$ of its current local root as $r(v)$, which we define as the id of the root node of the island which contains v. Initially by default the graph is connected, and consists of only one component (island) rooted at dst d, hence $r(v) = id(d) = 1$ for all nodes. Upon failure, v uses the value of its root and its neighbors to identify bridges, gateways, and islands. **Islands, Bridges, and Gateways Identification.** When node v loses its connectivity to its parent $p(v)$ (*i.e.*, the "on-tree" edge of $v \rightarrow p(v)$ fails), v becomes a new local root for its subtree(island), and triggers a `TopDownRootUpdate` procedure defined as following: $r(v) = id(v)$ if $p(v) = \phi$, otherwise $r(v) = r(p(v) \rightarrow v)$. Node v updates its children $c(v)$ along the DFS tree. Then the child node w propagates it to its own children, and so on. This process stops when it reaches the leaf nodes of the current island (subtree) which do not have any children nodes.

After a node sends its the new root local root $r(v)$ to its children, its ancestor neighbors and its descendant neighbors are still not aware (which are the possible candidates for bridges between islands). Hence, each node v needs to send its root information to them. The only difference is that these updates are sent across the "off-tree" edges only and are not propagated to other neighbors. With the local root information stored at each node, it is easy for a node v to discover whether it is a gateway or not, and whether an off-tree edge is a bridge or not.

5.5.2 Reachability Messages

The main purpose of these reachability notification messages is to let each local root know the minimum id it can reach addressing the different scenarios introduced in Fig. 5.3. Thus, this reachability information helps each island find this escape path if it exists. We now describe how these messages address the challenges mentioned in §5.4.2, the triggers to send them along with the guiding rules to avoid circular state information propagation.

If a subtree/island is still connected to the destination, its gateway nodes and bridges play a critical role in forming an "escape path" to the destination. These gateways need to inform their local root about the root and reachability information of the other remote island they are connected to. Hence, each gateway node g prepares a `GatewayUpdate` message to be sent to its local root node which contains $\{r(T_u), IR(T_u)\}$ where $r(T_u)$

indicates the root id $r(u) = id(u)$ of the remote island T_u where u is its local root, $IR(T_u)$ represents the Island Reachability information of island T_u which is initially equal to its root till T_u sends a new value. A gateway, can be connected to multiple islands, and/or multiple nodes in the same island. However, each gateway sends only one `GatewayUpdate` message which contains the information about all islands it is connected to. Upon the failure of any bridge, its corresponding gateways send `GatewayWithdraw` message to update the information at their local root nodes. We now describe the three rules that govern the propagation of reachability messages from one island to another.

[Rule I] Downward Local Minimal Root Discovery
Goal: To let each local root node v know the highest island (*i.e.*, smallest root) which it can reach either directly, or through its descendants.

Using archipelago (a) in Fig. 5.3 as example, initially island T_2 does not know that it is connected to the destination upon the failure of its on-tree edge. Hence, T_1 is required to inform T_2 that it can send its traffic to it through the bridge $(g_2^{(0)}, g_1^{(1)})$. The flow direction of the messages for Rule I is downwards, while the packets are sent upwards towards d. After the local root v receives the `GatewayUpdate` messages from all gateway nodes of its island, it can then classify its neighbor islands into: 1) ancestor islands $ANC(T_v)$: island T_u is an ancestor island of T_v w.r.to DFS tree if $r(T_u) < r(T_v)$, and 2) descendant islands $DES(T_v)$: island T_w is a descendant island of T_v w.r.to DFS tree if $r(T_w) > r(T_v)$. Then, each local root v can calculate its *ancestor reachability* (*i.e.*, min reachable remote island among its ancestor islands) as: $IR_A(v) = \min\{r(v), \min_{T_u \in ANC(T_v)} IR(T_u)\}$ where $IR(T_u)$ represents the island reachability of the ancestor island T_u, and u is its local root. Initially we set $IR_A(v) = r(v)$ if $ANC(T_v) = \phi$, or if node v has not received any information yet about its ancestor islands from its gateways. Then, each island T_v (including the mainland island which has the destination) recursively propagates its ancestor reachability value $IR_A(v)$ to its descendant islands $DES(T_v)$ downwards: $\forall T_w \in DES(T_v) : IR(T_v \to T_w) = IR_A(v)$. **[Termination Condition:]** This process stops when an island does not have any descendant islands (*i.e.*, the root of all its neighbor islands are smaller than its root $r(v)$). Each island T_v only sends its reachability value if it is less than its root $(IR_A(v) < r(v))$, otherwise this value is already known from the gateway connecting both islands.

[Rule II] Selective Upward Reachability Notification
Goal: for a descendant island T_w to help transfer the reachability information from its ancestor islands T_u to another ancestor island T_v if $r(T_u) < r(T_v)$.

For example archipelago (c) in Fig. 5.3, T_8 knows it can reach T_0. Hence, T_8 can send an upward reachability notification to island T_6 that its reachability is $IR(T_8) = 1$, and hence T_6 can use it to forward its traffic to reach the dst d because T_6 cannot reach it directly as the highest island it can reach upon failures is T_5 but it has reachability (v_5 ¿ 1). The question which arises now is: can the combination of [Rules I, II] introduce cycles of information and possible loops, we affirmatively answer that this is not possible because reachability info passed downwards during [Rule I] must be learned/obtained from ancestor islands only and hence the info T_6 learned from T_8 will never be propagated down to it, or any other descendant islands. Upon the end of [Rule 1] the local root v learns the reachability $IR(.)$ value of each ancestor island it is connected to through its gateways. Node v calculates its *overall reachability* from all remote islands as:
$IR(v) = \ min\{r(v), \min_{T_u \in ANC(T_v)} IR(T_u), \min_{T_w \in DES(T_v)} IR(T_w)\}.$
Initially $IR(T_w) = r(T_w)$ if node v has not received another value from T_w. Thus, if node v has a smaller reachability $IR(v)$ than what it learned from its ancestor island T_u, node v notifies the root node u about its reachability: $\forall T_u \in ANC(T_v)$ and $IR(T_u) > IR(v) : IR(v \to T_u) = IR(v)$. When the root node u receives this notification, it updates the reachability of its remote island T_v, it updates its own reachability $IR(u)$, and then checks its own ancestor islands $ANC(T_u)$ to check if any island is eligible to send this reachability information further upwards. **[Termination Condition]:** This process stops when an island does not have any ancestor islands (*i.e.*, the root of all its neighbor islands are larger than its root $r(v)$), or the ancestor island T_u has a smaller or equal reachability ($IR(T_u) \leq IR(T_v)$).

[Rule III] Conditional (Gated) Downward-Upward Reachability Range Notification

Goal: for a descendant island T_w to help transfer the reachability information from its ancestor islands T_u to another ancestor island T_v if $r(T_u) > r(T_v)$ (opposite case of Rule II).

In archipelago (c), during [Rule II] T_6 learns from T_7 that its reachability $IR(T_7) = r(T_5)$, but T_6 has a smaller reachability $IR(T_6) = 1 < IR(T_5)$ obtained through [Rule II] from T_8. This means that island T_7 is connected to another island T_5 which is not aware of the smaller reachability value of $IR(T_6) = 1$. However, [Rule I] prevents T_6 from forwarding its reachability to T_7 unless it was obtained through ancestor islands. At this point, either T_5 is disconnected from the dst d, or has not found its escape path yet. Since T_6 & T_5 are not directly connected, the reachability notification from T_6 has to go downwards first to T_7 which forwards it upwards to T_5. This notification is only valid for any island which has its root lies within this range $[IR_{min} = IR(v_6) = 1 : IR_{max} = IR(T_5) = r(v_5)]$. When T_7 receives this notification range, it marks that it can reach d through T_6, and forwards this notification to T_5 because its root lies within this range. Hence, T_5 marks that it can reach d through g_5, and there is no other island which satisfies this reachability range. Thus, this process stops here. We can notice that this rule is different from [Rule II] as the reachability information from (T_6) which has a higher root value $r(v_6) > r(v_5)$ to T_5 is opposite to the reachability flow direction of the combination of [Rules I, II].

To overcome the cycling of information, this is a conditional notification which has to be sent downwards to an intermediate island first which forwards it upwards to the target island(s) which have their root bounded by $[IR_{min} : IR_{max}]$. Both T_7, T_5 are not allowed to use this reachability value for reachability propagation outside this range, hence breaking the cycle. When a root w receives this island reachability range notification, it marks which gateway it received from to reach IR_{min}, and checks the reachability information all its ancestor and descendant islands which satisfy this range to forward this notification to them. **[Termination Condition]:** This process stops when a node receives this reachability range and does not have an island satisfying it.

At the end of these three rules, now each island can reach the destination as it has the proper reachability information. Upon failure, the island reachability information $IR(v)$ maintained at local root node v increases monotonically upon the convergence of the escape path search procedure.

5.5.3 Forwarding Rules

Forwarding Under Normal Operations

Under normal operations (*i.e.*, without any failures), upon receiving a packet p, node v can choose any ancestor node $u \in N_{anc}(v)$ to forward this packet towards the destination. For a more deterministic forwarding rule, v can pick $u^* \in N_{anc}(v)$ such that $id(u^*) = \min\limits_{u \in N_{anc}(v)} id(u)$, *i.e.*, the ancestor neighbor with the smallest id, thus using the off-tree edge if it exists. Another approach is to pick $u^* \in N_{anc}(v)$ such that $id(u^*) = \max\limits_{u \in N_{anc}(v)} id(u)$, *i.e.*, the ancestor neighbor with the largest id, thus the selected neighbor is the parent of each node in the DFS tree as it has the maximum id among all ancestors, and thus each node uses the on-tree edge of the original DFS tree. The advantage of using Off-tree edges is to jump some hops to reach the destination instead of following the DFST main branch on-tree edges. Following any of these rules, packets are guaranteed to reach d.

Forwarding Traffic Upon Failures

When node v loses its main nexthop, it constructs a `ReachabilityForwarding` table which maintains for each remote island T_u: 1) its root $r(u)$, 2) its reachability $IR(T_u)$, and 3) the id(s) of the gateway(s) connected to it, based on the reachability information received through Rules I, II, and III. Messages from gateways to the root v can be forwarded through their parent directly using the on-tree edges or an internal off-tree (*i.e.*, jump) edges to be faster if available. Based on the received reachability information, node v chooses to send its traffic to the island which provides the smallest reachability from that table, and hence a gateway connected to that island is selected to be the default gateway. In case of multiple gateways belonging to that remote island, or multiple islands providing the same reachability value, then the number of hops to the gateway or its id can be used as a tie-breaker. The final step is to activate/reverse links towards the selected gateway, hence the root node v reverses the direction of the edge which it received a message from the gateway through it, and similarly all the nodes along the path from v till the gateway reverse their edges as well, and update their nexthop towards the destination. Finally, the gateway activates the edge towards the selected remote island and forwards the traffic it receives for this dst towards this remote island. Having each affected island chooses another with a smaller reachability to forward its traffic upon failures is what guarantees this process converges and avoid forwarding loops as we prove next in details.

5.5.4 Resilient Routing Correctness

During the convergence process, we need to handle message exchange among nodes. These messages go downwards & upwards to reach other nodes and update their reachability information. Maintaining the following Invariants at all time at every node ensures that after executing all rules during the escape path search of the proposed resilient routing, each node is guaranteed to find an escape path to reach the destination as long as it is still connected to it, and this update procedure will converge. **Invariant 1:** The *id* value of a node v is always greater than or equal to its root's *id*. That is: $id(r(v)) \leq id(v)$. **Invariant 2:** The *id* value of a node v is always smaller than the id of its upper gateway (g) and greater than the id of its lower gateway (g'). That is: $gid(g|v) < id(v); gid(g|v) > id(v)$. **Invariant 3:** The island reachability of a node v based on reachability information from all ancestor neighbor islands is always smaller than its root's *id*. That is: $IR^*_A(v) < id(r(v))$

Invariant 3 implies and is to make sure that an island will send its traffic to another island with a lower root value, it can eventually reach the destination if the "escape path" exists. In the following sections we discuss the detail of rules we propose for the loop-free resilient routing to handle the messages exchanging and routing processes. One of the key properties behind the proposed protocol is that we can circumvent the "count-to-infinity" problem.

Theorem 5.5.4.1. *The correctness of the proposed loop-free resilient routing remains valid under the three rules of the escape path search procedure for handling multi-overlapping updates.*

The correctness of the proposed loop-free resilient routing is reflected via the following guarantees: i) determining escape paths (if exists) to reach the destination for nodes, ii) no circular reachability information propagation (reachability update procedure converges), and iii) avoiding the count-to-infinity problem.

Claim 1. *After the escape path search procedure has converged and assuming that there are no further failures, then $\forall\ v_i \in C_i$, $2 \leq i \leq c$, i.e., a disconnected component from destination d: $IR^*(v_i) = id(v_i)$, where v_i is the local root of the island/subtree with the minimum id in the disconnected component C_i, and for all the other islands/subtrees $w_i \in C_i$: $IR^*(w_i) \geq id(v_i)$. In this case, v_i knows it is now disconnected from the destination d, and it can immediately inform its children/descendants as well as its descendant islands. Thus, we can totally avoid the count-to-infinity problem.*

Proof of Claim 1. After the escape path search procedure has converged and v_i is the local minimal root of the island/subtree with the minimum id in the disconnected component C_i, it is obvious that $ANCI(v) = \emptyset$. Suppose $IR^*(v_i) < id(v_i)$, then we have either $r(v_i) < id(v_i)$ or $\min\limits_{I_w \in DESI(v_i)} IR^*(I_w|v_i)\} < id(v_i)$. If one of these possibilities occurs, then v_i is not the node with minimum id in the disconnected component C_i. This constitutes a contradiction. Note that the reachability of v_i, $r(v_i)$ is always less than or equal to its id ($id(v_i)$). Thus, $r(v_i) = id(v_i)$.

For all the other islands/subtrees $w_i \in C_i$, we use the same argument in above proofs. We have: $IR^*(w_i) = \min\{r(w_i), \min\limits_{I_p \in ANCI(w_i)} IR^*(I_p|w_i), \min\limits_{I_q \in DESI(w_i)} IR^*(I_q|w_i)\}$. Suppose $IR^*(w_i) < id(v_i)$, it means:

- $r^(w_i) < id(v_i)$, or
- $\min\{r(w_i), \min\limits_{I_p \in ANCI(w_i)} IR^*(I_p|w_i) < id(v_i)$, or
- $\min\limits_{I_q \in DESI(w_i)} IR^*(I_q|w_i)\} < id(v_i)$.

If one of above cases occurs, v_i then is not the node with the minimum id in the disconnected component C_i. This constitutes a contradiction and therefore, $IR^*(w_i) \geq id(v_i)$. ■

Claim 2. *After executing the three rules of the escape path search: For $v \in V$, $v \neq d$, v will find an escape path to reach the destination and this update procedure converges.*

Proof of Claim 2. For ease of exposition, we consider each island T_i (sub-tree) as a *super node*, let's hereafter call, node T_i (*e.g.*, node T_1, T_2, T_3, T_4, so on) as depicted in Fig. 5.3. We now show that after executing the three rules of the escape path search, each node can find an escape path to reach the destination (if exists) by considering the network under three cases:

Straight path: connects a node T_i to the destination through $ANS(T_i)$. This case is depicted in Fig. 5.3a in which T_2 can reach the destination through T_1 ($T_2 \rightarrow T_1 \rightarrow d$). This can be easily done by using Rule I in which each node (island) recursively propagates its reachability to its descendant downward.

None-straight path: wherein a node's traffic has to go downward through its descendant and then go upward through its ancestor to reach the destination. To illustrate how the traffic is handled in this case, consider again the simple toy example in Fig. 5.3b with the node T_3 must go downward through T_4 before going upward to reach the destination d. The question becomes how the node T_3 figure out this path. Recall that after the Rule I, every node knows the minimum reachability from their ancestors. Using the Rule II, then nodes that learn more than one different reachability information will send the better one to its ancestors those have a greater reachability. By this way, T_3 can find the escape path and reach destination.

Interleaving path: that includes interleaving of many downward and upward subpaths. The path that connects T_{11} to the destination depicted in Fig. 5.3d demonstrates this case ($T_{11} \rightarrow T_{10} \rightarrow T_{12} \rightarrow T_9 \rightarrow T_{13} \rightarrow T_{14} \rightarrow d$). The question is that how to discover this path for the T_{11}. After the Rule I and Rule II, every node learns the reachability from its ancestors and descendants. However, if a node v receives the reachability information from one of its descendants (w) and the reachability value provided by w is greater than its current reachability, it means w is connected to u that is not aware of the smaller reachability value of v ($IR^*(v)$). In this case, using the Rule III, node v will send its reachability information down to its descendant (w) that is connecting to another node u with a greater reachability value. After receiving the information from node v, w will forward the reachability information to node u. Hence, u can be aware of the reachability of v. Thus, u can find the escape path and reach the destination. ■

Claim 3. *Using the three rules of the escape path search ensures there is no circular reachability information propagation.*

Proof of Claim 3. Without loss of generality, let's consider each island as a node. In fact, there are different forms of the topology that may cause the circular reachability information propagation. However, the key point behind these different forms is the same wherein the information that a node v has learned from its ancestor and propagated downward to its descendant node, is sent back to its ancestor by its descendant via another branch. However, using the Rule I and II, this cannot happen. In the Rule I, each node v (an island with the local minimal root v) leans the reachability $IR^{*}(.|v)$ value of each its ancestor u. Likewise, node w, an descendant of v is also received the reachability information. In the Rule II, before node v forwards upward the information to u ($\forall u \in ANCI(u)$), it always checks to see whether u is eligible to receive the reachability information that v learned from its descendants w or not. The information is forward upward to u if only if the reachability that node v learned from w is smaller than what it learned from its ancestor. That is why the reachability that w has received from v will NOT be forwarded upward to node u, a v's ancestor. Therefore, there is no circular reachability information propagation when using the three rules of the escape path search. ■

The previous claims are leveraged by the conditional reachability propagation used in the update procedure which is defined as the principle of **"Restricted & Conditional Propagation of Reachability Information"**: i) The minimal reachability learned by a gateway from nodes *within* the same island will be *conditionally* propagated across different islands. ii) Likewise, the minimal reachability learned by a gateway *from another gateway of another island* will be *conditionally* propagated *within* the island it belongs to. Moreover, it is also worth indicating that the proposed resilient routing can guarantee no forwarding loop, if does, these loops are only temporary.

Claim 4. *Under normal operations, the above simple forwarding rule guarantees there is no forwarding loops.*

Proof of Claim 4. It is obvious that because the packet of any node v is always forwarded upward to a node in $N_A(v)$, then under the normal operation, the above simple forwarding rule guarantees there is no forwarding loops. ■

Claim 5. *Upon failures, if $y \in ANC(v)$ i.e., y is an ancestor of v (not necessarily a neighbor of v i.e., y may not be in $N_A(v)$), and y is connected to the destination d without using v (i.e., both $v, y \in C_1$). Then, y should never reverse the edges towards v to find a path to the destination d, nor does any of y's descendants u along the path from v to y.*

Proof of Claim 5. For any node u belongs to any path from v to y, there always exists at least a path connecting u to y. In addition, because y is connected to the destination d without using v, u always can reach destination via y. Likewise, for any node x connected to u either directly or indirectly, can reach the destination via u. Therefore, y should never reverse the edges towards v to find a path to the destination d. ■

Claim 6. *As long as node v is still connected to the destination d, the previous forwarding rules will find this path.*

Proof of Claim 6. First of all, it is sufficient to show that if there exists the path connecting an island to the destination, then all nodes connected either directly or indirectly can use our rules to find this path. That is because once the local minimal root of the island realizes the escape path, all nodes inside the island can find down the way to direct their traffic to the gateway on the escape path, and finally their traffic can be conveyed to the destination. Let's consider each island as a node. After the Rule I, II and III, every node (island) knows about the reachability of all its ancestor and descendants. The reachability information of the node connected to the destination does not changes and this information is propagated in the way designed in the Rule I, II, and III to all its neighboring nodes (islands). Therefore, all nodes in the network finally will find the existing path by using the Rule I, II and III. ■

Claim 7. *During any step of the update processes of the reachability information upon failures, some nodes may take some forwarding decisions based on stale information which may lead to forwarding loops, but these loops are temporary. After all update processes have converged, with no more failures, the above forwarding rules leads to no forwarding loops.*

Proof of Claim 7. Suppose after all update processes have converged, and there are no further failures, there exists the forwarding loops in the network. Without loss of generality, we assume that after failures and all update processes have converged node y's traffic is forwarded in a loop. It means there are two possibilities: i) y has lost its connection to the node which has the value of the updated min reachability, or ii) y *did not* receive the updated reachability information triggered by affected (by failures) nodes. Notice that the designed mechanism is used for the connected component only, there always exists at least a path connecting any node to the destination. In addition, after the Rule I, II, and III, every node is updated about the reachability of all its ancestors and descendants. Likewise, its ancestors and descendants has min reachability to reach the destination. If either possibility above happens, it means that y *has not* receive the updated reachability information, and therefore the update processes have not converged. This constitutes a contradiction. Thus, after all update processes have converged, and there are no further failures, there is no any forwarding loop in the network. ■

5.6 Implementation

We have implemented our resilient routing protocol in NS-3 [81]. Our implementation includes: a) control plane operations, and b) data plane operations. The controller first computes the DFS tree for each destination d and assigns a unique node identifier (id) for each node during the DFS construction based on its visit order as mentioned earlier. During the bootstrap phase, the controller initializes the state information for each node: its id, its list of neighbors, its parent, and children along with their respective ids. This process is executed only once at the beginning. Based on this pre-stored initial information, each node can then classify its neighbors as ancestors and descendants using their ids as mentioned earlier. Then, each node uses the forwarding rules stated in §5.5.3 to specify its main nexthop and use it to initialize its the routing table. Other links to the ancestors can also be used, however initially they are *InActive* waiting to be utilized upon failures to reach the destination.

Upon link failures affecting both ends, each node starts its procedure to decide if any changes are required. If the node does not lose its main nexthop, then its forwarding operation is not affected. For example if the failed link was connecting a node to its ancestor or descendant, it needs to update its local state while continuing to forward packets as they do not depend on updating its local state. Upon the failure of a node's link to its parent, it invokes the *CreateNewIsland* procedure using its id as the new root value. The goal of this procedure is to discover the alternatives ancestors connected to it directly or through its descendants to find its new nexthop using its local state information and its descendants' information. Each node sends its new root to its neighbors through the heartbeat messages. Upon receiving this information, each node learns whether it is a gateway or not. Root nodes of different islands exchange *reachability* information through the gateways connecting them as described in §5.5.2. Once the affected root node finds a gateway to forward the traffic of the affected island, the nodes on the path from the root to this gateway update their nexthop and routing tables pointing towards the gateway. Other nodes in the island remain unaffected and as well as other nodes which are not part of this island.

After the exchange of *reachability* information among neighboring islands, if the root nodes does not have any connection to a neighboring island with a lower root value, it is now disconnected from the destination. This phase can be transient during reachability information propagation or permanent if it lost connection to the destination. However, our resilient routing algorithm is capable of quickly detecting and avoiding *count-to-infinity* problem by identifying a node as an articulation point during the bootstrap phase. An articulation node v is defined as following: if v loses its connection to all its ancestors upon failures, the whole subtree (*island*) rooted at v is now disconnected from the destination. Then, we can very quickly prevent it by having node v declare itself as dead, and send this dead flag to its descendants which will do the same thing. Hence, the whole subtree rooted at v will mark itself dead, and stop any useless trials to reach the destination d. This is one of the main advantages of DFS over BFS. In DFS, any *off-tree* edge connects a node to its ancestors only and there are no cross-edges between different subtrees at the same level compared to BFS. Thus, the detection of the articulation point is much easier in DFS. Finally, to minimize the number of messages sent between nodes, some destinations affected by the same link failure and require the same link reversal can be grouped together and sent as one message for the list of affected destinations. These messages can be piggybacked in data packets in response to link/node failure and recovery events, or sent as separate packets if there are no data packets.

5.7 Evaluation

In this section, we highlight the issues with existing path-based protocols, and then compare the performance of our resilient routing protocol with DDC [70][6] as a recent representative of link reversal protocols utilizing DAG. Although DDC is also robust against k arbitrary link/node failures as long as the network is not partitioned, it employs link reversals to reconstruct a new DAG whenever any node other than the destination becomes sink (*i.e.*, no outgoing link). In contrast, our resilient routing attempts to find the least number of changes required by affected nodes to restore connectivity to the dst.

5.7.1 Experimental Setup

We evaluate our proposed resilient routing protocol and DDC using NS-3 simulator on AS topologies from RocketFuel [82] which vary in number of nodes and links. The simulator takes as input the topology, and a set of links to fail (chosen at random from the total links for each topology). We fail one link at a time, upon the detection of failure both protocols converge, then a new failure is introduced based on the new converged state of each, and so on. We compare them w.r.t. the number of link reversals required as well as the nodes participating in the convergence process. However, we first show the impact of failures with existing path-based protection routing schemes which shows the need to effectively leverage the topological diversity inherent in the network to maintain connectivity between source (src) and destination (dst) nodes under various failure scenarios as proposed by our routing algorithm.

5.7.2 Path-based Approaches

For each src-dst pair in a given topology, we consider the following path protection and link protection techniques: the primary path (the shortest path (d) between this src-dst

[6]We obtained the code from the authors.

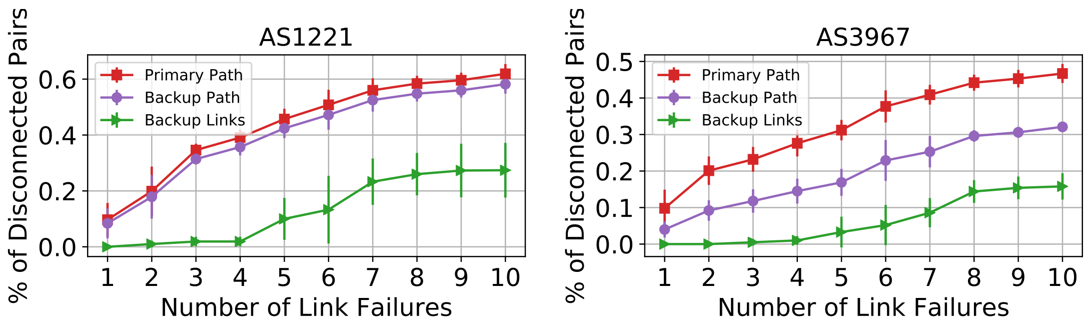


Figure 5.4: Percentage of Disconnected Pairs

pair) is protected via i) a *backup path* which is calculated as the next shortest path after removing the primary path from the topology, or ii) a set of *backup paths* per link – for each link, we find an alternative disjoint path to the nexthop after removing this link.

Disconnection. Using AS1221 (83 nodes, 131 edges), and AS3967 (91 nodes, 180 edges) as examples, we calculate the percentage of disconnected pairs (*i.e.*, no path between src and dst) among all src-dst pairs in the topology to measure how each technique utilizes the topological diversity. For each routing scheme $\mathcal{R}$, a src-dst pair is considered *disconnected* with respect to $\mathcal{R}$ if the failed links render all pre-computed (primary or backup) paths/rules *invalid*, *i.e.*, packets from s can no longer be routed to d if the failed edges lie on the shortest path for primary path technique, lie on both the shortest path and its alternative disjoint path for backup path, or lie on any link of the shortest path and its backup disjoint path for backup links technique. From Fig. 5.4, we see that the percentage of disconnected pairs increases with increasing the number of failures using existing path and link protection routing schemes even though the src & dst are still connected. Hence, a resilient routing with the capability of effectively leveraging the topological diversity is required to maintain connectivity under arbitrary link failures. Our resilient routing starts with one main nexthop for each node, but upon failures additional links are activated/reversed as long as the src-dst pair are still connected.

Convergence Time. Using AS1221, Table 5.1 shows the convergence time for our resilient routing algorithm, which differs from one dst to another, hence we show both the median and maximum times across all destinations in msec. We can notice that with

Table 5.1: RR Convergence Time for AS1221

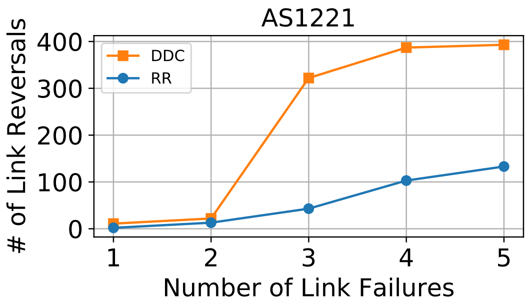
Num of Link Failures	**1**	**2**	**3**	**4**	**5**
Median Time (msec)	0	2	2	4	5
Max Time (msec)	1	8	16	21	36

5 link failures and a convergence operations after each link failure, the total maximum time in less than 36 msec involving 75 link reversals. This is compared to path-based protocols which upon each link failure require state updates among all nodes followed by route recomputation and updating the routing tables which usually takes 100s ms or more to converge.

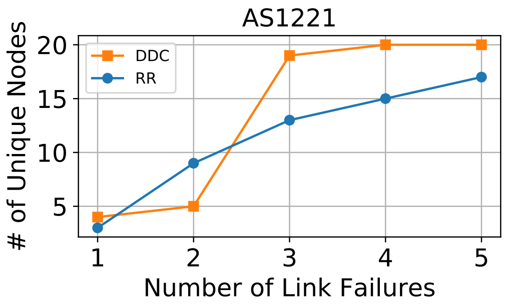
5.7.3 Link Reversal Approaches

Link Reversals & Unique Nodes Fig. 5.5a shows that compared to DDC, our resilient routing requires much lower number of link reversals and updates to the routing tables even with increasing the number of failures. The is due to only nodes affected by failures need to reverse a link towards another node to restore connectivity to dst. However, in case of DDC, some (or all) links are reversed for every sink node until the only sink node is the dst, which leads to many nodes updating their nexthop and routing table even more than once as the reversals happen through packet headers. This also results in a lower number of nodes involved in the convergence process as shown in Fig. 5.5b.

Cost. Table 5.2 shows the number of reachability update messages between root nodes and gateways, or between root nodes across multiple island exchanged during convergence, which again differs w.r.t. dst. Since at each node the same outgoing links can be shared by multiple dst nodes, these messages can be aggregated if they are directed towards the same node. We can see that some link failures incur zero reachability messages, and on average with 5 link failures and multiple convergence processes a total of 19 messages were sent per dst and a max of 84, which is still far away from the update messages sent via link state or distance vector algorithms during convergence.



(a) Link Reversals



(b) Unique Nodes

Figure 5.5: Number of Link Reversals and Unique Nodes

Table 5.2: RR Update Messages for AS1221

Num of Link Failures	**1**	**2**	**3**	**4**	**5**
Median # of msgs	0	8	8	13	19
Max # of msgs	2	10	23	37	84

5.8 Summary

In this chapter, we have developed a novel *resilient* routing algorithm that is guaranteed to find a path, if available, from a source node s from a destination node d after an arbitrary number of link failures, *without resort to route recomputations.* This is thus, to the best of our knowledge, the *first provably correct* proactive routing algorithm that is resilient against arbitrary network failures. Our algorithm is based on *depth first search* (DFS) spanning trees and intelligently exploits several of its crucial properties to route around failures. We pre-compute a set of (per-destination) *monotonic* node identifiers (*id*'s) which captures the partial ordering induced by the DFS tree and encodes network connectivity information intelligently. These node *id*'s are pre-installed (together with appropriate forwarding rules) at network nodes to guide them for loop-free packet forwarding by maintaining reachability under failures. In particular, we have devised multi-stage process for *localized, conditioned* message exchanges among neighboring nodes to quickly filter and select eligible next-hops for packet forwarding. We have implemented the algorithm using NS-3, and conducted comparative evaluations based on simulations. The evaluation results demonstrate the efficacy of our solution and its superiority over existing routing schemes under failures.

Our solution decouples ensuring *reachability* from (routing) performance, and thereby makes it possible to quickly find alternative "next-hops" for reachability under arbitrary failures. This is in contrast to conventional/existing (resilient) routing schemes often couple the two concerns; as such, they are unable to maintain reachability under arbitrary failures. In this work we have emphasized exclusively on discovering and maintaining reachability under arbitrary failures. However, we can readily augment our solution with additional (routing) metrics (*e.g.*, in terms of bandwidth, latency or other QoS metrics) to combine the concerns with discovering/maintaining reachability with finding and forwarding using "best" possible paths in terms of a given performance metric – *e.g.*, routing along a shortest latency path as much as possible while routing around failures. This, for example, can be achieved by running a separate performance metric update protocol which can then be used to select the "best" nexthop among multiple *eligible* ones, while always ensuring connectivity/reachability. For example, one may employ the Bellman-Ford algorithm for (*local*) update of the routing performance metrics, where we no longer need to be concerned with the "count-to-infinity" problem as the routing metrics update messages will only be exchanged among nodes in a still connected island or archipelago. Likewise, one may invoke a (localized) link state update process within a (connected) island to compute the shortest paths to the eligible gateways (to the mainland) after failures.

Chapter 6

Mobile Video Streaming Using 5G Cellular Network

6.1 Introduction

Over the past decade, the video delivery has shifted from traditional TV broadcasting to online video streaming due to the growth of several video-on-demand services such as YouTube, Netflix, Hulu, ... etc, and the continuous growth of smartphone users which is anticipated to exceed 50% of the total mobile devices by 2022 [83]. This lead the Internet traffic to be dominated by video streaming. In 2020, mobile video streaming represented 65% of the global mobile downstream traffic [2], and is expected to reach 79% in 2022 [3]. Another ongoing trend is Head-Mounted Displays (HMDs) such as Oculus Rift and Google Cardboard which are available for the public. Connecting smartphones to these HMDs, users can watch 360° which are now available on YouTube and Facebook platforms. However, this demands a very high bandwidth for streaming to provide smooth high quality of experience (QoE) to users without freezing and rebuffering events.

Due to the need for a much higher speed and lower latency, the 5th generation (5G) wireless technology made its debut as commercial services to costumers in early summer 2019. `Verizon` launched the world's first commercial millimeter wave (mmWave) 5G in Chicago and Minneapolis. This was followed by the three other major U.S. carriers (`T-Mobile`, `Sprint`, and `AT&T`), and many other carriers around the world. Most 5G

Table 6.1: 5G Technologies Adopted by Major U.S. Carriers

Carrier	**Verizon** [84]	**AT&T** [85]	**T-Mobile** [86]	**Sprint** [87]
Type	mmWave	mmWave	mmWave	mid-band
Freq.	28/39 GHz	24/39 GHz	28/39 GHz	2.5 GHz

deployments employ mmWave technology which promises to achieve a throughput up to 20 Gbps which is 100× better than today's 4G [12] as well as lower latency. Hence, 5G has a potential to support several emerging bandwidth-hungry multi-media applications such as ultra-HD (UHD) 4K/8K, 360° and volumetric (AR/VR) video streaming, as well as low-latency cloud gaming, and vehicle-to-everything (V2X) communication.

The performance of mmWave technology is achieved by a series of innovations in massive MIMO, advanced channel coding, and scalable modulation, and the higher frequency at which mmWave operates from 24 GHz to 53 GHz with abundant free spectrum. However, due to its short wavelength, mmWave signals are highly directional, require line of sight (LoS), and thus are very sensitive to the surroundings and vulnerable to obstacles. Thus, mmWave has a limited coverage, and require dense deployments of 5G towers and require complex beamforming algorithms to overcome these issues. Another alternative deployment approach is to use mid-band frequencies (1 - 6 GHz) which provides a larger coverage than high-band, but lower bandwidth capacity. This approach was adopted by `Sprint`, however despite the issues of mmWave it was used by the other three U.S. carriers. Table 6.1 lists the frequencies[1] at which each U.S. carrier operates. Supporting such a wide range of frequency spectrum range required a complete overhaul of the 5G radio. Thus, 5G radio is also referred to as 5G-NR (NR stands for New Radio).

At the time of this study, there were no available public datasets or tools to reflect and capture the real performance of these commercial 5G services. Hence, we built our own measurement tool and methodology to provide an impression about the performance of these services. We studied `Verizon`'s mmWave 5G in Minneapolis city due to its anticipated high bandwidth compared to mid-band deployments. Through several experiments with different settings, we found that mmWave 5G can in-deed offer ultra-high bandwidth (up to 2 Gbps) compared to less than 300 Mbps for 4G. However, there

[1]Based on the 5G services provided by the carriers as of October 2019.

are many practical issues with 5G, even with clear LoS, 5G throughput exhibits much higher variation than 4G, mainly due to the PHY-layer nature of mmWave signal which can be easily blocked. This is even more severe when the user is moving, because of the frequent handoffs incurred due to obstructions caused by nearby buildings, vehicles, and other factors. Thus, the UE loses connection to 5G and switches to 4G, and the throughput even drops to zero (5G "dead zones"). For example, we experienced a total of 31 handoffs while walking a short 700 m loop. This high amount of switching may confuse applications as it it is very hard for applications to cope with which may lead to highly inconsistent user experiences.

The study is centered around the following fundamental problem: *How can we endow bandwidth-intensive applications with the abilities to fully take advantage of the (potential) ultra-high bandwidth offered by (mmWave) 5G while at the same time overcome its highly variable throughput performance so as to deliver good and consistent quality-of-experience (QoE) to mobile users?* To address this fundamental challenge, we use mobile *volumetric video streaming* as a case study. Such application requires bandwidth as high as 750 Mbps. Using mmWave 5G throughput traces, we first conduct trace-driven simulations (see §6.5) to answer the following two basic questions: 1) are (volumetric) video streaming applications equipped with existing *adaptive bitrate* (ABR) algorithms ready to take advantage of 5G's high throughput? and 2) how does the wild throughput fluctuations affect the application performance from the perspective of QoE (measured in terms of video stall times)? Our investigation reveals that wild fluctuations in 5G throughput often lead to quick buffer depletion under poor channel conditions, especially when entering 5G "dead zones," thereby resulting in a large stall time that has a significant impact on user's QoE. Our findings illustrate that new mechanisms are needed to endow applications with the abilities to fully utilize the potential of 5G while overcoming its challenges. We refer to applications endowed with such capabilities as *being 5G-aware.*

We advocate new mechanisms to make applications *5G-aware* (§6.6). We first note that ABR algorithms used in existing video streaming applications rely mostly on *in-situ* bandwidth "probing" for throughput estimation. The highly variable throughput performance of mmWave 5G, coupled with frequent handoffs, make such methods ineffectual [88]. We argue that a) more sophisticated *machine learning (ML) methods for* *effective throughput prediction*[2] that can account for diverse environmental factors and be able to forecast 5G throughput over a longer time horizon are needed to aid applications in intelligent bitrate adaptation. Furthermore, we advocate b) *adaptive content bursting* – namely, employing (significantly) larger buffers (both at the client side as well as within the 5G radio network) – to allow the 5G radio network to fully utilize the available radio resources under good channel/beam conditions to burst as much content as needed to the client so as to prepare for and bridge over the 5G bandwidth troughs and dead zones. In addition, c) employing *dynamic radio (band) switching* (*e.g.*, between 5G and 4G or between 5G high, mid, and low bands) is crucial in maintaining session connectivity and ensuring minimal bitrates.

We conduct trace-driven experiments (§6.7) to evaluate the efficacy of these strategies in overcoming the wild fluctuations of 5G throughput performance. We identify both the opportunities and challenges offered by emerging 5G services, and call for new mechanisms to make applications *5G-aware* – namely, enabling applications to take full advantages of opportunities offered by 5G while overcoming the new challenges it poses. Our experimental results demonstrate that these strategies can consistently deliver high video quality (compared to the theoretical optimal performance), and in particular, minimize, and even completely eliminate video stall times, despite 5G dead zones. Our study clearly constitutes only an initial step towards this direction – much more work needs to be done by the research community to make applications *5G-aware*.

In the following sections, we introduce the background about 5G deployment, and related work in §6.2. In §6.3, we present our measurement tool. In §6.4, we discuss the characteristics of 5G throughput and we show the performance of video streaming under 5G in §6.5. Our proposed mechanisms to build 5G-aware applications are presented in in §6.6, to overcome the 5G throughput fluctuation and dead zones. Finally, we present our evaluation in §6.7, identify key research directions on improving 5G users' experience in §6.8, and the Chapter is summarized in §6.9.

[2]In [89] we have demonstrated that it is feasible to predict (mmWave) 5G throughput using machine learning algorithms with weighted average F1 score of above 0.95. Such high accuracy is shown to be adequate for video ABR adaptation [90].

6.2 Background and Related Work

5G Deployment. 5G can be deployed by either a Non-Standalone Deployment (NSA) or a Standalone Deployment (SA) [91]. In case of NSA, 5G-NR is used for data plane operations, while the control plane operations still go through the 4G LTE infrastructure. In case of SA, both the data plane and control plane operations are performed through 5G infrastructure totally independent from the 4G LTE cellular infrastructure. For rapid and cost effective deployment, carriers deploy their first commercial 5G services by co-locating 5G network equipment with the existing 4G LTE infrastructure. This led to having a high bandwidth as mentioned which can reach (up to 2 Gbps), however the lower latency was not achieved due to the NSA deployment.

mmWave. Several studies have been conducted on mmWave deployments in indoor and outdoor environments [92, 93, 94, 95, 96, 97]. mmWave signals are directional and highly sensitive to the surroundings, and hence can be blocked by several materials including human body, tree foliage, trains, and tinted glass. To overcome this issue, complicated algorithms have been developed for beamforming and reflections as indicated by these beam tracking studies [98, 99, 100]. However, none of these studies have studied the performance of mmWave in real commercial 5G service on smartphones.

5G Throughput Traces. In order to analyze the performance of video streaming applications using 5G networks, we need 5G throughput traces. We investigated the following options in order:

- Option 1 Using public datasets: however, at the time of this study there were no publicly available 5G throughput measurement datasets conducted in the wild for the recently launched commercial 5G service. We are also not aware of any dedicated public monitoring tool for 5G networks.
- Option 2 Run tools like `tcpdump` on a device with 5G capability: the only available commercial of-the-shelf devices with 5G capability from `Verizon` in Summer 2019 are Samsung Galaxy S10 5G (SM-G977U) referred to as SGS10, which have a built-in 5G radio for accessing 5G. Running `tcpdump` requires root privilege of devices which was difficult for SGS10.
- Option 3 Build our own measurement tool (described in §6.3).

Several studies have been conducted on mmWave deployments from theoretical point of view [92, 93, 94, 95, 96, 97], however,our work [101] was the first measurement study on the performance of commercial 5G services by different US carriers. Using 5G traces, the authors in [88] illustrate why current video streaming ABR algorithms do not work well with 5G mmWave. One of the main reasons is attributed to the inaccurate 5G throughput estimation, as was also shown by Zou *et al.* in [102] that better throughput prediction can indeed improve the video performance in cellular networks. Lumos5G [89] was the first ML model to predict 5G throughput with high accuracy illustrating the inefficiency of existing 3G/4G throughput prediction ML-based and data models which can only rely on user location [103, 104]. These studies further support our argument for the need to build robust 5G ML throughput prediction models in video streaming apps as well as the need for new mechanisms to make them 5G-aware.

Volumetric video streaming is a hot topic which has been recently investigated. For example [105] proposes a manifest file format for volumetric video streaming following the DASH standard. Nebula [106] utilizes edge servers to decode the 3D data and generates a 2D video instead. ViVo [107] applies visibility-aware optimizations to enable real-time streaming. These techniques are complementary to our work and can be integrated with our proposed strategies. Other research studies focus on evaluating the QoE performance for video streaming using simulated 5G traces such as [108, 109]. To the best of our knowledge, our work is the first to study the issues in using commercial mmWave 5G for volumetric video streaming using real-world 5G throughput traces, and propose new mechanisms to build 5G-aware applications.

6.3 5G Measurement Tool

As mentioned before, the only option to monitor the 5G service and measure throughput was to build our own tool to obtain the 5G throughput traces. Another challenge we faced while building the tool is how to obtain useful information about 5G such as whether we are currently connected or not, 5G tower information, and signal strength. At the time of this study, the state-of-the-art Android OS (version 10) claims to provide access to 5G-NR related APIs [110, 111, 112]. We identified some locations for 5G service area using `Verizon`'s coverage maps [113]. However, we found that `Verizon` does

Table 6.2: 5G Connection Status

Variables	Description
`nrAvailable=FALSE`, `nrStatus=NONE`	UE is not in a 5G service area
`nrAvailable=TRUE`, `nrStatus=NOT_RESTRICTED`	UE is in a 5G service area, connected to 4G
`nrAvailable=TRUE`, `nrStatus=CONNECTED`	UE is in a 5G service area, connected to 5G

not provide any meaningful responses to these APIs, and hence they are useless. For example, we find that when the phone is connected to 5G, the `getDataNetworkType()` API of Android `TelephonyManager` still returns LTE. Thus, we had to investigated the raw fields of several objects searching for useful information. We found that the raw string representation of Android's `ServiceState` object contains two important fields: `nrAvailable` and `nrState`. Hence, we logged these fields every second on our user equipment (UE) SGS10. After the analysis of these two fields, and with repeated experiments, we found the values are: `nrAvailable=TRUE or False`, `nrStatus=NONE or NOT_RESTRICTED or CONNECTED`. We found with a very high confidence that combination of the values of these two fields indicate the 5G connection status as summarized in Table 6.2. The reason why the UE can be in a 5G service area but connected to 4G is either due to a poor 5G signal, or the 5G signal is blocked as mentioned before that the 5G signal is highly sensitive to the surrounding environment and can be blocked by several materials such as human body, train, ... etc.

To measure the throughput, we cross compiled `iPerf` 3.7 [114] and embed it into our measurement tool to issue one or more `TCP` connections to periodically download data from a backend server for bandwidth probing. This integration allows us to monitor the performance of 5G throughput under different settings such as geolocations, mobility mode, number of `TCP` connections, ... etc. Overall, the fields collected by our measurement tool are listed in Table 6.3, and Fig. 6.1 is a screen shot from our tool.

Finally, to ensure our measurements are not affected by any device artifacts or bottlenecks, we performed the following steps:

1. We purchased multiple SGS10 devices and used our measurement tool to measure the 5G performance from these devices at the same time and location to ensure that our experiments are not affected by any device artifacts. We confirmed that they exhibit similar 5G performance.

Table 6.3: Fields Recorded by Our 5G Measurement Tool

Field	Description
`timestamp`	logs the date and time every second
`latitude`, `longitude`	UE's fine-grained geolocation
`mCid`	the cell ID of the tower the UE is currently connected to, parsed from the raw `ServiceState` object
`userActivity`	either walking, still, driving, ... etc using Google's Activity Recognition API
`throughput`	downlink speed reported by `iPerf` 3.7
`5GServiceStatus`	calculated as discussed in Table 6.2

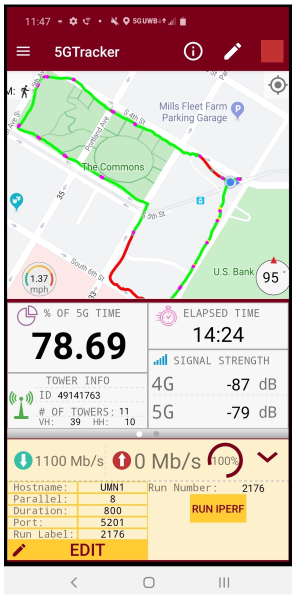


Figure 6.1: 5GTracker Tool

2. We also ensured that the device-side is not a bottleneck because SGS10 is a high-end smartphone equipped with an octa-core CPU, 8 GB memory, Qualcomm Snapdragon 855 System-on-Chip (SoC), and X50 5G modem. Thus, it can handle 5G's high throughput which can reach up to 2 Gbps.

3. We used the same device SGS10 to compare the throughput performance of 4G and 5G since SGS10 supports both networks.

4. We performed our experiments repeatedly under different settings (such as mobility mode, locations, time of day, ... etc) to ensure our 5G throughput are representative and not biased.

5. We conducted our experiments using `iPerf` 3.7 servers, and we found that the server geographical location and cloud service provider affect 5G performance. Hence, we used the following criteria to select servers to use to make sure that the bottleneck of the path between the UE and `iPerf` 3.7 server is not the Internet.

   (a) Downloading from these servers yields highest 5G throughput (statistically) compared to servers in other locations and/or providers.

   (b) Using other wired (non-mobile) hosts yields at least 3 Gbps throughput, well beyond the 5G peak speeds.

6. We verified that the difference between throughput measured by our tool and from the commercial Ookla Speedtest [115] service is less than 5%.

6.4 5G Throughput Characteristics

Using our measurement tool, we conduct experiments at several locations in Minneapolis to understand the characteristics of 5G throughput. We first start by identifying visually multiple 5G towers, and denote their tower ids (`mCid`) using our measurement tool.

For each experiment we: i) make sure we have a line of sight (LoS) between the phone held in hand and the panel, ii) stand about 25-30 meters from the tower, and iii) make sure the UE is connected to 5G (`5GServiceStatus`). Then, we start a `TCP` bulk download for 60 seconds, and measure the throughput every second (reported by `iPerf`). We change the experiment settings to have {1, 2, 4, 8, 16} parallel `TCP` connections, and repeat each experiment at least 3 times at each location. All experiments are done under clear weather to make sure the weather does not have any impact on our results. With these steps, we believe our results represent realistic measurements of the performance of urban 5G access from smartphones. For fairness, we use the same settings and SGS10 device to perform the same tests over 4G to compare its performance and characteristics with 5G.

For all bulk download tests, we consider the throughput measurements in our results after 20 seconds of the `TCP` flows start to discard the impact of `TCP` slow start on our measurements. We show the measurement results for 4G and 5G for different number of concurrent `TCP` connections in Fig. 6.2. Each box plot represents all 1-second measurement samples collected for a specific setup. We make the following observations.

1) 5G vs. 4G. We can notice that with clear LoS, 5G offers much higher throughput than 4G, for example using 8 parallel `TCP` connections, the median for 5G and 4G throughput are 1467 and 167 Mbps, respectively, which represents more than 8x improvement in performance. Hence, 5G has potential to support bandwidth-hungry applications which was not available before. However, despite the presence of LoS, 5G throughput exhibits much higher variations than 4G. This could be attributed to the

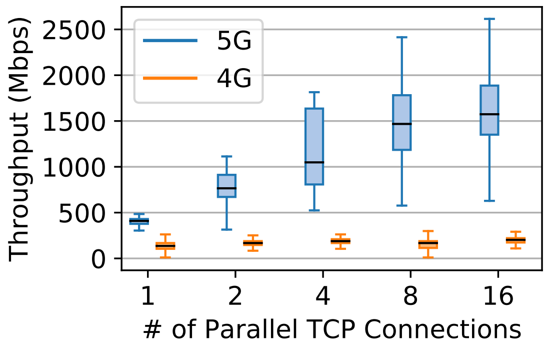


Figure 6.2: 5G Throughput Under Stationary LoS

physical layer nature of 5G signals as well as potential inefficiencies at various layers. For example, at PHY/MAC layers, smartphones' small form factor makes engineering a 5G modem challenging [116]. At the transport layer, an excessive number of `TCP` connections may incur cross-connection contentions, which may also lead to throughput variation in particular for 5G whose available bandwidth is less stable compared to 4G.

2) Impact of TCP Concurrency. We can also notice that 5G throughput improves as the `TCP` concurrency level increases, with the bandwidth being fully utilized when there are more than 8 concurrent connections. Through controlled experiments, we confirm that this is 5G-specific, *i.e.*, it does not appear in wired, WiFi, or 4G networks. It is either because the 5G carriers are imposing per-TCP-connection rate limiting, or because they are not able to support very high throughput for a single `TCP` connection. This will affect the performance of single-connection protocols such as `HTTP/2` [117] which can not fully utilize the available 5G bandwidth. Hence, application developers are encouraged to aggressively increase the `TCP` concurrency in their applications. Otherwise, using one `TCP` connection may adversely affect the performance of their application operating in 5G networks.

3) Impact of Mobility. In order to study the impact of mobility on 5G performance, we choose an area with a 5G deployment and walk with a speed of ≈ 1.4 m/s. We use two SGS10 devices held side by side to compare the performance of 4G and 5G. We start a `TCP` bulk download for 500 seconds and measure the throughput for both networks. The collected traces are shown in Fig. 6.3. We can notice that 5G throughput

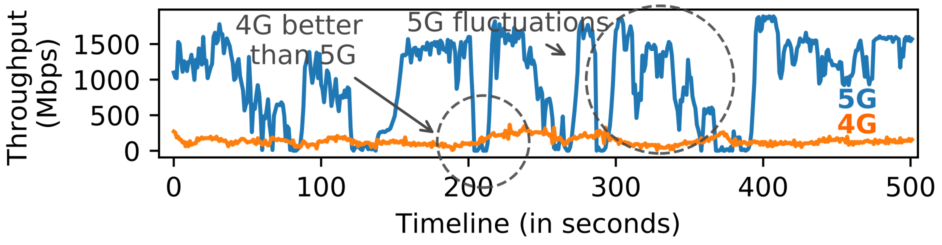


Figure 6.3: 4G and 5G Throughput Traces while Walking

fluctuates wildly from one location to another when compared to 4G. 5G throughput can reach as highly as 2 Gbps, but sometimes it can quickly drop below that of 4G, and even to nearly zero. We notice that in some regions we do not get 5G connection and the UE falls back to 4G network due to non line of sight (NLoS) with any 5G tower. NLoS happens due to blocking the 5G signal by tree foliage, moving bodies *e.g.*, train which can't be penetrated by the 5G signals, and the absence of surrounding buildings with reflecting surfaces to reform the 5G beams. We remark that the large, wild fluctuations of 5G throughput and dead zones would pose severe challenges to existing video streaming applications as we show next.

6.5 Video Streaming Performance Under 5G Throughput

As shown by our measurement that mmWave 5G offers new opportunities to enable several emerging bandwidth-hungry applications, but it also poses many challenges. In this section, we use volumetric video streaming application as a case study to illustrate the performance of this class of applications under 5G throughput.

Volumetric videos[3] differ from regular and 360° videos in that they are truly 3D, with each frame consisting of a 3D point cloud. During playback, a user wearing a MR (mixed reality) headset can freely navigate herself with six degrees of freedom (6 DoF) movement, gaining an immersive telepresence experience. A volumetric video can have for example 350K points per frame played at 30 frames per second (FPS). Each point takes 9 bytes (3 bytes for RGB color and 6 bytes for its 3D location). This yields a total of 350K$\times$30$\times$9$\times$8 $=$ 756 Mbps *when uncompressed.* While the 756 Mbps throughput

[3]A sample of a high-quality volumetric video streaming can be found at `https://www.youtube.com/watch?v=feGGKasvamg`.

requirement far exceeds the capacity of the existing 4G LTE service, it is well within the ultra-high bandwidth offered by the commercial mmWave 5G service under our study.

Unfortunately, decoding (*compressed*) point cloud data requires heavy-weight algorithms such as *octree* [118, 119, 120] that cannot be effectively supported by today's mobile phones at the 30 FPS frame rate [121]. Thus, streaming uncompressed volumetric videos to mobile phones is the only practical solution at the moment. Moreover, streaming *uncompressed* volumetric videos also makes it easier to adopt a flexible, *layered* approach for video bitrate adaptation: starting with a base layer (which imposes a minimum bandwidth requirement), we can dynamically adapt to the available network bandwidth by adjusting the resolution (*i.e.*, increasing or decreasing the number of points) of an entire (or portions of) 3D video frame.

To understand the impact of the large, wild fluctuations of 5G throughput on existing video streaming applications, we used the 5G trace from Fig. 6.3 to stream a volumetric video for 500 seconds played at a constant rate of 350K points per frame (see §6.7.1 for more details about experiment settings). This resulted in a total stall time of 90 sec out of the 500 seconds experienced by 17.92% of the frames of the video. A stall occurs for every missing frame at its playback time till the frame is downloaded from the server. Despite the very high throughput of 5G, the reason for this "non-smooth" quality-of-experience (QoE) to users with frequent stalls is attributed to the sudden and quick drop in 5G throughput. Also, existing video streaming applications do not take full-advantage of the extra available throughput (which can reach 2 Gbps) which will end up being wasted. This is indicated in Fig. 6.4 which shows the maximum number of frames in the buffer corresponds to 4.2 secs (*i.e.*, 126 frames) which are not enough to cover long 5G dead zones. Only, when the network throughput varies more or less "smoothly", client-side buffering would work reasonably well and help further "smooth out" the effects of short-term throughput fluctuations, which is not the case for 5G.

To understand the impact of the large, wild fluctuations of 5G throughput on existing video streaming applications, we use the 5G trace from Fig. 6.3[4] as a representative trace to stream a volumetric video for 500 seconds played at a constant rate of 350K points per frame (see §6.7.1 for experiment settings). We measure the performance by total

[4]Throughout this chapter, we use mobility traces to study and overcome the impact of 5G dead zones, on top of 5G throughput variability which occurs for stationary users. These user mobility scenarios are likely to happen for Autonomous Vehicle applications and vehicle to everything (V2X) technology.

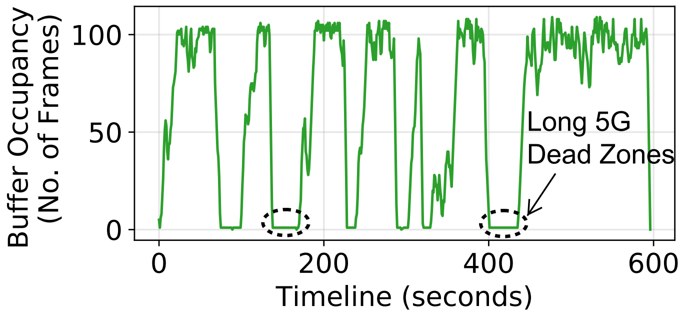


Figure 6.4: Buffer Occupancy During a 500 sec Video Using 5G Throughput Shown in Fig. 6.3.

stall time; a stall (rebuffering) occurs for every missing frame at its playback time till the frame is downloaded from the server. This results in a total stall time of 90 seconds (18%).

Despite the very high throughput of 5G, this "non-smooth" QoE to users with frequent stalls is attributed to the sudden and quick drop in 5G throughput. Also, existing video streaming applications do not take full-advantage of the extra available throughput (that can reach as high as 2 Gbps) thus might end up being wasted. This is indicated in Fig. 6.4 which shows that the maximum number of frames at any point in the buffer corresponds to 4.2 secs (*i.e.*, 126 frames) which are not enough to cover long 5G dead zones which can extend to 20 secs. Only, when the network throughput varies "smoothly", client-side buffering would work reasonably well and help further "smooth out" the effects of short-term throughput fluctuations, which clearly is not the case for mmWave 5G. This raises the questions of i) how long the buffer should be to cover 5G dead zones, and ii) which bitrate quality to request as it affects the time and bandwidth required to download each frame.

The bitrate is often determined by the estimated throughput. However, traditional bandwidth estimation approaches which rely on the short-term past history and use methods like harmonic mean or other methods (*e.g.*, [122]) are not adequate for 5G throughput due to its wild and non-smooth variation. Moreover, 3G/4G networks can rely on location to predict the cellular performance [103, 104], however mmWave 5G throughput is more complicated as it is affected by multiple factors and is very sensitive to the surrounding environment. Hence, traditional location-based prediction models are insufficient.

This trace-driven simulation points out both the opportunities and challenges in mmWave 5G, and shows that existing video streaming applications do not work well over mmWave 5G. Hence, we need to rethink about the way these applications are built to become 5G-aware. There is a need to come up with novel mechanisms to effectively utilize the extra high bandwidth offered by 5G whenever available while at the same time coping with the wild fluctuations and occasional “dead zones” to improve the user’s QoE.

6.6 Building 5G-Aware Video Streaming Apps

In this section, we propose new mechanisms to make bandwidth-intensive applications *5G-aware* so as to take full advantage of 5G networks while overcoming their new challenges. First, we highlight the need for new ML throughput prediction mechanisms, then put forth several *cross-layer* mechanisms to effectively utilize the available radio resources and improve user’s QoE despite 5G’s high throughput variability and dead zones.

6.6.1 Need for ML 5G Throughput Prediction

Despite the wild variability of 5G throughput compared to 4G, our recent study [89] argues, through extensive experiments and statistical analysis, that by controlling the key user-side (UE) factors affecting 5G, the throughput can largely be characterized and can be predictable. These key factors include for example user’s geolocation, mobility mode, mobility speed, and user’s compass direction. Then, it proposes *Lumos5G* – a composable machine learning framework which considers different combinations of contextual and environmental factors, and applies the state-of-the-art machine learning algorithms for making context-aware 5G throughput predictions with a higher accuracy over existing traditional prediction methods. As an example, Fig. 6.5 shows the distribution (or spread) of variation seen in 5G throughput traces (aggregated using 40 runs collected over a span of 20 days) along a walking route: the dark center curve represents the average throughput and shaded areas represent the 25% to 75% percentile range. From this figure, we can notice that there are some patches when the throughput is consistently high, while others the throughput is consistently low. Although not shown,

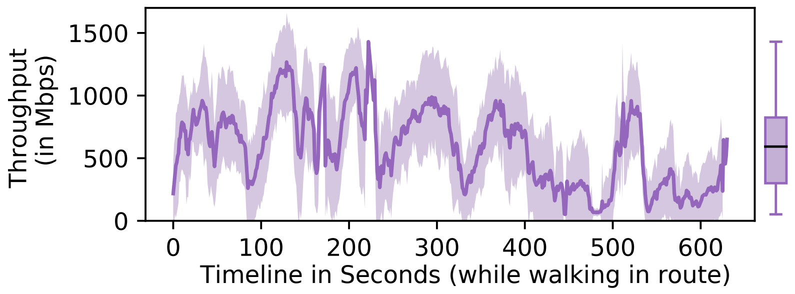


Figure 6.5: Variation in 5G Throughput.

we also observe that the throughput characteristics and variation drastically vary when the user is walking in the opposite direction. This signifies the importance of compass direction as a key factor in characterizing 5G throughput.

Ideally these ML models can be deployed at 5G base stations, users can collect the UE key factors, and report them to the 5G base station to train the ML models. In return, the user receives a bandwidth prediction map containing 5G dead zones (with a start position and a length) as well as the current/future throughput prediction over a longer time horizon for different routes[5]. With the ability to predict the near future 5G performance in/around the current user's location, video streaming applications can then make intelligent decisions to download video frames as explained next to provide exceptional QoE while at the same time adapt smoothly to 5G's high variation and fluctuations. Additionally, these throughput prediction models can also be used by cellular networks themselves for adaptive beam forming, resource allocation, preemptive handoffs, and improving network coverage.

6.6.2 Adaptive Streaming Mechanisms

We put forth several mechanisms to enable applications to fully take advantage of ultra-high bandwidth afforded by (mmWave) 5G while also mitigate the impact of high throughput variability due to fast varying frequency radio bands.

• **Adaptive Content Bursting.** The goal of this mechanism is two-fold: 1) to "burst" sufficient amount of application data to the 5G radio network so that the 5G radio resource control sub-layer can fully take advantage of available radio resources whenever

[5]See [89] for more details about the bandwidth prediction maps and ML deployment.

possible, *e.g.*, when a clear LoS path or good quality high-frequency channel is available; and 2) to bridge over 5G low-bandwidth troughs and “dead zones” by delivering as much data as needed to a user/UE when the channel conditions are good. Goal 1) requires provisioning larger buffer at the radio network, and is motivated by the fact that radio resource allocation and transmission scheduling are often based on the amount of per-user data in the radio network buffer. If a high-quality radio channel or LoS beam is available to a UE but there is little data in the per-user buffer, the 5G radio network cannot fully take advantage of the ultra-high bandwidth offered by 5G. Ensuring there is always sufficient data in the per-user buffer via adaptive content bursting will avoid such “lost opportunities”. Goal 2) entails allocating larger buffer at the UE/client side. Clearly, for both to work effectively, the ability to predict channel conditions and (future) 5G throughput, *e.g.*, based on the user orientation, mobility and environmental factors, with ML techniques, is crucial, so that the amount of burst data can be dynamically adapted to balance buffer requirement, QoE, and radio resource utilization.

• **Dynamic Radio Switching.** Through our extensive experiments, we find that in some patches while UE is connected to 5G (but with poor channel quality), 4G in fact yields a higher throughput (see Fig. 6.3). In other times, UE may enter a 5G dead zone while still under 4G coverage. Hence *proactively* switching between 5G and 4G based on estimated/predicted channel conditions or throughput performance will be crucial in maintaining connectivity and ensuring a minimal bitrate, especially during user mobility. Likewise, dynamically switching between diverse radio channels/bands is also essential in coping with diverse and fast varying channel characteristics (*e.g.*, bandwidth, bit error rate).

In a nutshell, we believe that combining these new (cross-layer) mechanisms, coupled with effective ML-based throughput prediction, will be the key to enable a new class of bandwidth-intensive applications such as volumetric video streaming. Incorporating these new mechanisms entails re-designing the adaptive bitrate (ABR) and other algorithms used in existing video streaming applications so that they can fully utilize the ultra-high bandwidth and other capabilities afforded by (mmWave) 5G, while also help them mitigate various PHY-layer challenges posed by mmWave 5G radio – in other words, making them *5G-aware.*

6.6.3 Theoretical Bounds for Choosing Video Quality

Increasing the video quality (number of points for each frame) leads to increasing the stall time if the current network conditions can not support the requested quality as the client's buffer would not be able to maintain a threshold number of frames. Thus, selecting the appropriate quality given the network conditions is crucial as it impacts the user's QoE. We attempt to answer this question by considering an *ideal* case where we have *perfect knowledge* of the available 5G network throughput over a period of time, and derive theoretical bounds on the best video quality we can achieve without any stalls.

Suppose we start streaming a video of length T seconds at time t_{start}. With a start delay of d seconds, the playback begins at $t_1 = t_{start} + d$, and ends at $t_{end} = t_1 + T$. Let F be the frame rate (*e.g.*, $F = 30$); $n = T * F$ is the total number of frames to be played, with a rate of one frame played every $1/F$ seconds. (We will use τ_k, $k = 1, ..., n$, to denote the playback time of the kth frame, where $\tau_1 = t_1$ and $\tau_n = t_{end}$.) Given a trace of available 5G bandwidth from t_{start} to t_{end} (see Fig. 6.3 for example), we are interested in finding out what is the *best achievable video quality* Q defined as the *highest constant* (thus the "smoothest") bitrate *without any stalls*. We obtain the following theorem for the upper- and lower-bound of Q using content bursting to fully utilize the available bandwidth.

Theorem 6.6.3.1. *Given a trace of (instantaneous) network throughput rate* $b(t)$, $t_{start} \leq t \leq t_{end} (= t_{start} + d + T)$, *let* $B(t) = \int_{t_{start}}^{t} b(t)dt$. *Then the highest achievable constant bitrate without any stall is given by* $Q_* \leq Q \leq Q^*$, *where* $Q_* = \min_{1 \leq k \leq n} B(\tau_k)/k$ *and* $Q^* = B(t_{end})/n$, *where* $n = T * F$.

We remark that in the statement of the theorem, we are ignoring the network latency (and round trip delays) between a mobile client and a video streaming server. We are essentially assuming that this latency is negligible, *e.g.*, when the video streaming server is located in a mobile edge cloud (*e.g.*, co-located with the cell towers) very close to the mobile user. With a non-negligible network latency λ, we need to subtract λ and use, *e.g.*, $t'_{end} = t_{end} - \lambda$, in the statement of the theorem so as to ensure the last bit of a kth video frame has arrived at the mobile client side before its scheduled playback time τ_k.

Proof of Theorem 6.6.3.1. The proof is illustrated in Fig. 6.6, where we have plotted

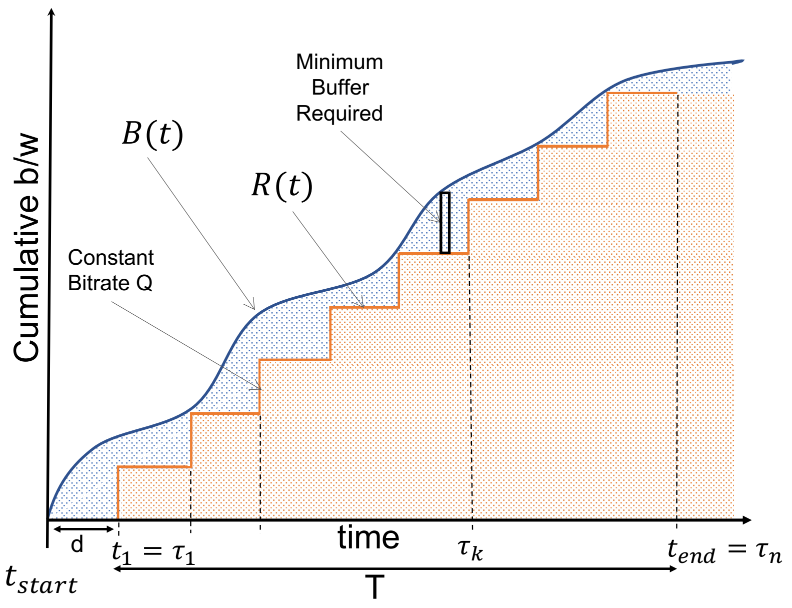


Figure 6.6: Illustration of the Proof

the cumulative throughput $B(t)$,
$t_{start} \leq t \leq t_{end}$ $(= t_{start} + d + T)$ as a function of time. Note that given a constant bitrate video of quality Q, namely, each frame contains Q bits, the total amount of bandwidth required for the video delivery at this level is $Q * n$, where n is the total number of frames in the video. Since $B(t_{end})$ is the maximum cumulative network bandwidth available between times t_{start} and $t_{end} = \tau_n$, the maximal video quality achievable is at most $Q^* = B(t_{end})/n = B(\tau_n)/n$. More generally, we note that by τ_k (the playback time at the kth frame, at least $Q * k$ amount of data must have been delivered to the client in order for the client player not to stall. In other words, we must have $B(\tau_k) \geq Q * k$. It is not hard to see the minimal video quality level we can achieve *with no stalls* is given by $Q_* = \min_{1 \leq k \leq n} B(\tau_k)/k$. The minimum buffer size required to avoid stalls while serving frames with quality Q is $buf = \max_{t_{start} \leq t \leq t_{end}} (B(t) - R(t))$ which represents the maximum difference between the two curves $B(t), R(t)$. ■

When dynamic 5G/4G switching is employed along with content bursting, this is equivalent to using a modified network throughput trace $\bar{b}(t)$, $t_{start} \leq t \leq t_{end}$ which uses the maximum value of the 5G throughput and the 4G throughput. The theoretical bounds can then be obtained via Theorem 6.6.3.1 with $\{\bar{b}(t)\}$.

6.7 Evaluation

In this section, we conduct trace-driven experiments to demonstrate the benefits of these mechanisms. In particular, we investigate how effectively adaptive content bursting will allow the 5G network to fully take advantage of ultra-high bandwidth when available and help the application to bridge over 5G bandwidth troughs and dead zones. We will also use the real-world 5G/4G throughput traces we have collected to emulate *dynamic radio (band) switching* (between 5G and 4G) to examine its potential benefits in maintaining session connectivity and in further enhancing the user's QoE. These mechanisms will be aided by ML-based 5G throughput prediction [89]. We will in particular prioritize video stall times, and compare the results obtained with the theoretical bounds on the best video quality achieved without any stalls.

6.7.1 Experimental Setup

Currently there is no way to do radio(band) switching, hence, we built our own emulated video player, using the TCP/IP protocol stack and C++, to fetch video frames from the server to show its effectiveness using real 5G commercial traces. The client player has a large playback buffer (virtually unlimited) to ensure our emulation's performance metrics are able to reflect the network's performance as opposed to the device's hardware specifications. Using our measurement tool, we have collected 4G and 5G traces simultaneously 3 times every day for more than 20 days using Samsung Galaxy S10 5G devices while walking in a dense 5G deployment area in downtown Minneapolis for Verizon's NSA 5G Service. These traces share a common behavior as shown in Fig. 6.5, hence we pick a representative 5G & 4G network traces shown in Fig. 6.3 captured during our study while the user is walking at a speed of $\approx$ 1.4 m/s, and replay it using `tc` [123] to throttle the bandwidth to match the 4G and 5G throughput. We use BBR as TCP congestion control algorithm developed by Google to reduce the impact of TCP slow start due to wild fluctuations. In these experiments, we request frames using constant bitrate[6] (*i.e.*, all frames are requested with the same number of points

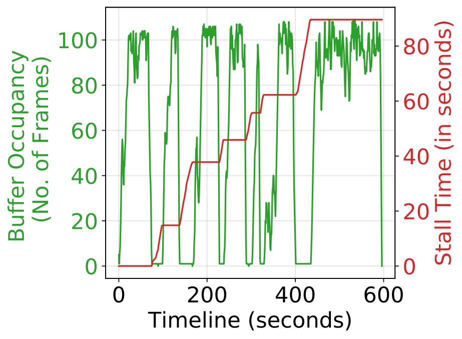
[6]See §6.8 for variable bitrate quality.

per frame 350K), and we use the *stall time* (*i.e.*, rebuffering duration) as a metric for user's QoE. We emulate watching the video using 3 modes: *1) 5G Only*: by only using the 5G throughput trace shown in Fig. 6.3. *2) Dynamic 5G/4G Switching*: with the bandwidth estimation knowledge, the player proactively switches between 4G and 5G networks depending on which one has the higher available bandwidth. *3) Content Bursting + Dynamic Switching*: in addition to the dynamic switching, the video player also proactively bursts future content as much as possible when extra high bandwidth is available as estimated by the bandwidth estimation module to handle the 5G dead zones shown in Fig. 6.3. We emulated a 500 seconds video requested at 350K points per frame for these modes, each experiment was repeated at least 3 times with minimal differences among runs, hence a representative run from each mode is shown in Fig. 6.7 for buffer occupancy and stall time.

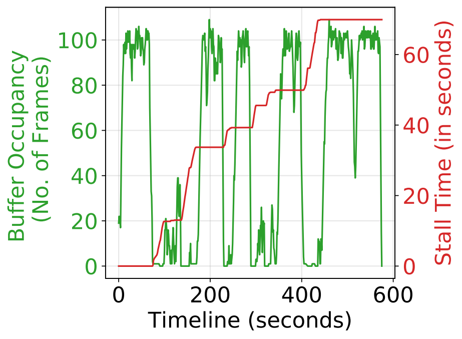
6.7.2 Experimental Results

A) Buffer Occupancy and Stall Time. *1) 5G Only* mode: Fig. 6.7a shows the user experiences a large stall time of around 90 secs (out of 8-min walk) with 17.92% of the video frames experiencing stalls. This is due to having a maximum throughput of 200 Mbps in 5G dead zones which is not enough to receive and play frames of 350K points which require a total of $350\text{K}\times30\times9\times8 = 756$ Mbps. Thus, the user has to wait till they pass these dead zones and get back 5G connectivity to resume fetching frames. Also, the buffer occupancy never exceeds 126 (*i.e.*, a playback length of 4.2 secs) which is clearly not enough to cover 5G dead zones which have longer duration. *2) Dynamic Switching* mode: Fig. 6.7b shows that with the bandwidth estimation knowledge, switching to 4G shields 5G dead zones reducing the stall time to 70 secs experienced by 14.04% of the video frames. This is attributed to 4G's omnidirectional radio which helps maintain the basic data connectivity during mobility. *3) Content Bursting + Dynamic Switching* mode: Fig. 6.7c shows when the client player utilizes the ultra-high bandwidth of 5G to proactively request additional frames from the server, the stall time is reduced to 21 secs but was not completely eliminated. However, we can notice that the maximum buffer occupancy increased to 724 frames which helped overcome some 5G dead zones but not all.

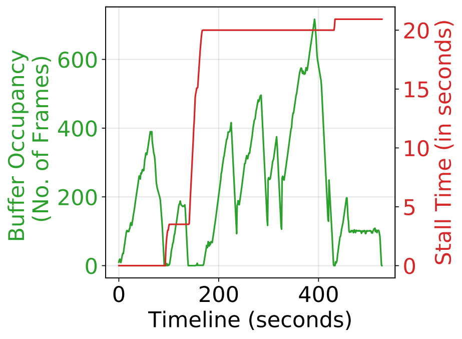
B) Selecting Appropriate Bitrate. Applying Theorem 6.6.3.1, to the given trace



(a) 5G Only



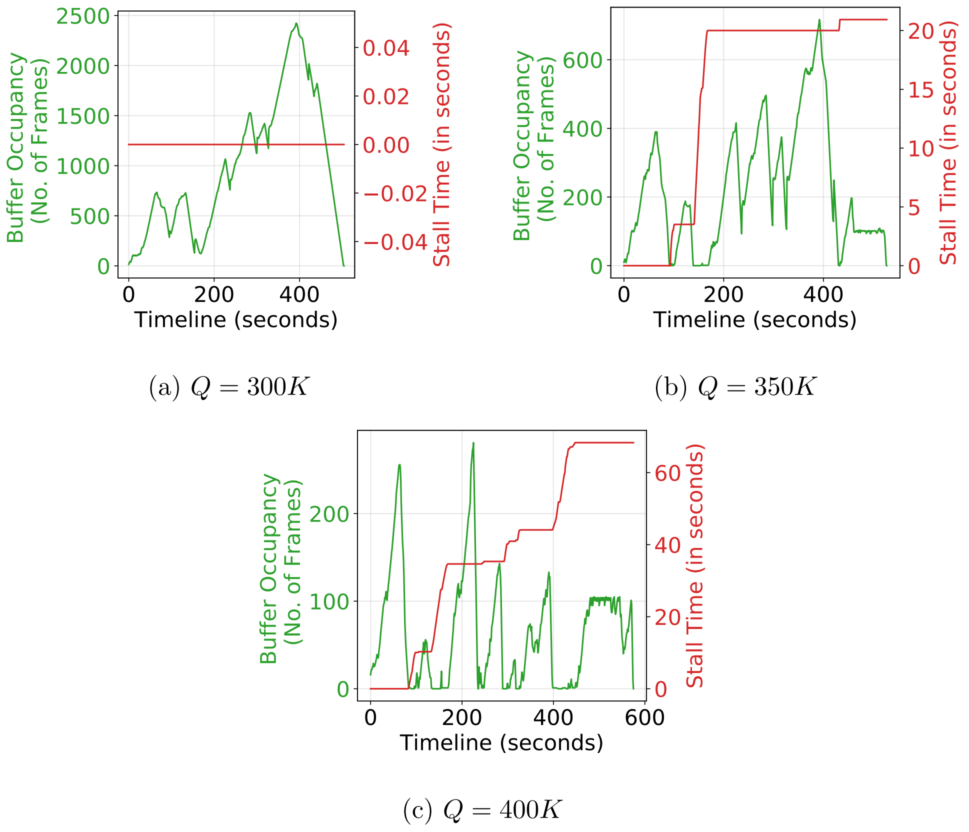
(b) Dynamic Switching



(c) Content Bursting + Dyn. Switching

Figure 6.7: Buffer Occupancy and Stall Time During a 500 SEC Video Streamed With Quality 350K.

in Fig. 6.3, we found that requesting frames using the video quality at 300K points eliminates any stalls, while other higher video qualities always result in a stall time. Hence, we repeated the previous experiments by streaming the video using a quality of 300K points per frame with *Content Bursting + Dynamic Switching* mode. The stall time was completely eliminated while maintaining the full frame quality overcoming the throughput fluctuation and dead zones in the 5G throughput trace. Fig. 6.8 shows the buffer size for the different video qualities when *Content Bursting + Dynamic Switching* mode is employed. We noticed that when the video quality increases, the buffer takes more time to build and consequently gets depleted quickly before/at the dead zones increasing the stall time even when *Content Bursting + Dynamic Switching* mode is employed. The reason for this behavior is that requesting a bitrate higher than what



(a) $Q = 300K$

(b) $Q = 350K$

(c) $Q = 400K$

Figure 6.8: Buffer Occupancy and Stall Time during a 500 sec video Streamed with different Qualities using Content Bursting + Dynamic Switching Mode

can be supported by the available bandwidth prevents the buffer from building up as it requires more time to download each frame. Table 6.4 summarizes the stall time for the different modes and video qualities.

C) Radio Time for 4G and 5G. We use the time spent using each radio (4G/5G) shown in Fig. 6.9 as a simplified representation for the consumed energy during streaming the video using 300K points per frame. When *Dynamic 5G/4G Switching* mode is used, 4G is enabled for a limited time when its throughput is higher than 5G, and the stall time is minimized to 52 secs and hence 5G radio time decreased. Using *Content Bursting + Dynamic 5G/4G Switching* leads to completely eliminating the stall time, and both radios were ON for the shortest time.

It is worth mentioning that the above proposed mechanisms are proof-of-concept.

Table 6.4: Stall Time for Video Playback.

Points/Frame	**300K**	**350K**	**400K**
Required Throughput	648 Mbps	756 Mbps	864 Mbps
5G Only	82 sec.	90 sec.	106 sec.
Dynamic 5G/4G Switching	52 sec.	70 sec.	79 sec.
Content Bursting + Dynamic Switching	0 sec.	21 sec.	68 sec.

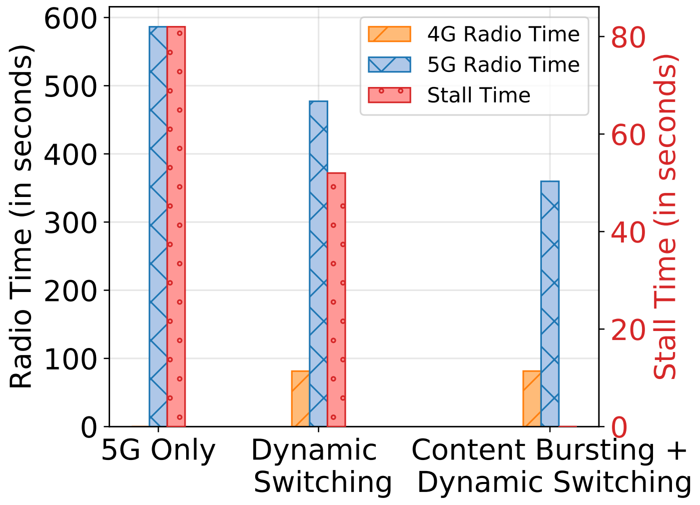


Figure 6.9: Radio Time for 4G and 5G During a 500 SEC Video Streamed With Quality 300K.

We envision that additional mechanisms with more judicious usage of Content Bursting and Dynamic Switching using ML throughput prediction will bring further performance improvements for video streaming applications under challenging 5G network conditions as we discuss in the next section.

6.8 Discussion

In this section, we elaborate on future directions for video streaming applications to further enhance their performance.

- **Scalable Video Coding (SVC).** Most video players use advanced video coding

(H.264/MPEG-4 AVC) standardized in 2003 [124] which encodes a video frame into different bitrate versions independently of each other leading to redundant information. A major drawback in AVC encoding is that it cannot adapt to the high fluctuations of 5G bandwidth. Thus, another alternative encoding Scalable Video Coding (SVC) was developed which is an extension to H.264 standardized in 2007 [125]. In SVC, a frame is encoded in a base layer (lowest quality), and multiple enhancement layers which can be used to improve the quality in an incremental way. For each frame, if the base layer is missing at the playback time, a stall will occur; if the higher-quality enhancement layers are missing but not the base layer, the frame will be played at a low quality to avoid stalls; if all layers are present, the frame will be played at the original (highest) quality. This resolves the wasted bandwidth problem of AVC by using layering technique and hence can just download the additional layers up to the specified quality level. SVC comes at the cost of decoding overheads at the client, however nowadays hardware decoders using GPU are available in smart phones.

• **Streaming Variable Quality Levels.** A video can be delivered using either: i) a constant bitrate level which requests all frames with the same quality (*i.e.*, same number of points per frame); or ii) variable bitrate levels in which the video player switches between different quality levels for different frames. This decision depends on the network condition, its variability, and the buffer occupancy. Thus, instead of using the minimum constant bitrate level to avoid stalls as defined by Theorem 6.6.3.1, the video bitrate level can change over time according to the predicted throughput with the goal of eliminating stalls while maintaining video quality smoothness (*i.e.*, avoid bitrate fluctuations which degrade user's QoE). For example, when the user mobility mode (still, walking, driving) changes, the mobility speed affects the available bandwidth. Hence, instead of prefetching frames with a very high quality, a more judicious decision can be made based on the predicted future bandwidth to decide which quality to use to avoid stalls. Thus, avoid requesting frames with the highest quality which yields only few frames in the buffer that will be depleted quickly. The goal is to develop an adaptive algorithm which can avoid stalls while at the same time deliver the highest possible quality with smooth quality variation instead of frequent changes from the highest quality to the lowest quality.

Theorem 6.6.3.1 not only demonstrates how to obtain bounds on achievable best video qualities, but also hints on how we may perform adaptive bitrate (ABR) selection for achieving best video qualities given bandwidth prediction for the upcoming X seconds. At the current time t, given the predicted network bandwidth $\tilde{b}(t)$ over $(t, t+\Delta t]$. Using the predicted total available bandwidth $B(t, t+\Delta t) = \int_t^{t+\Delta t} \tilde{b}(t)dt$, we employ Theorem 6.6.3.1 to determine the best video qualities for the next $\Delta k = \Delta t * F$ frames to be fetched. To account for uncertainty in the bandwidth prediction, a more conservative approach can be followed to assign priorities (or "deadlines") for fetching (future) content of different qualities: by prioritizing using the current (stable) available bandwidth to burst lower qualities of future Δk frames first than using it to increase the qualities of more recent frames. This will ensure a minimal video quality to users with *no* stalls while "smoothly" adapting to higher qualities whenever possible. This illustrates the power and utility of ML bandwidth prediction in enabling new mechanisms for 5G-aware applications to utilize the ultra-high bandwidth of 5G and overcome its wild fluctuation and dead zones.

• **Multi-Band Aggregation.** 5G supports a broad and diverse range of frequency spectrum. The low-band frequency provides maximum coverage but limited bandwidth, while high-band provides very high bandwidth but its signals are highly sensitive and vulnerable to obstacles thus limiting its coverage. Between both these extremes lies the mid-band range, which provides higher bandwidth capacity than low-band & better coverage than high-band. Since the debut of commercial 5G deployments, carriers supported a single class of frequency range. While high-band (mmWave) range can provide very high bandwidth capacity, its suffers from limited coverage. Hence, several carriers now consider deploying multiple classes to leverage multiple frequency bands which is known as *multi-band 5G*, enabling carriers to aggregate multiple channels to achieve higher data rates. In such situations, low-band and mid-band 5G will allow carriers to provide stable 5G service with wider coverage, while offering mmWave 5G to support bandwidth-heavy applications [126, 127, 128]. Multi-band 5G is now also supported by 5G chip manufacturers who have developed a single-chip which supports multi-band, *e.g.*, Qualcomm's Snapdragon X55 5G modem-RF supports both mmWave and sub-6 GHz 5G new radio [129]. Streaming *uncompressed* volumetric videos makes it easier to adopt a flexible, *layered* approach for multi-band 5G deployment and video bitrate adaptation. Low-band and reliable radio channels with good conditions can be used to stream the base layer with the minimum video quality & bandwidth requirement, while *simultaneously* mid-band/high-band 5G are used to stream higher quality enhancement layers by dynamically adapting to the available network bandwidth through adjusting the resolution (*i.e.*, increasing or decreasing the number of points) of an entire (or portions of) 3D video frame.

• **Cross-layer Design.** Due to the new challenges posed by 5G, we believe cross-layer mechanisms are required to improve user's QoE such as *e.g.*, dynamic radio resource allocation (see [130] for discussion), PHY-layer/MAC-Layer/RRC-Layer info passed to the transport layer so that congestion control (CC) algorithms can work well. For example due to frequent handoffs in mmWave 5G, packet loss might affect the congestion window (cwnd). If signal strength improves and if we know it is going to be stable, then we might want to increase the cwnd sooner than following the CC algorithm approach which might under-utilize the available bandwidth.

6.9 Summary

This preliminary study points out both the opportunities and challenges in mmWave 5G, and shows that existing video streaming applications do not work well over mmWave 5G. We proposed new mechanisms "content bursting" and "multi-radio switching" to make video streaming applications 5G-aware, and derived theoretical bounds for choosing the appropriate bitrate to avoid stalls during video delivery. We managed to show that with more judicious usage of 5G/4G and appropriate bitrate selection, we can eliminate the stall time while streaming a video with constant bitrate improving the application performance under 5G's challenging network conditions. We have also discussed some key research ideas to improve the user's QoE even more. Therefore, we need to shift the way we develop applications for 5G to take advantage of its high throughput and at the same time be able to tackle its challenges and cope up with its huge throughput variations and frequent handoffs when the user is moving.

Chapter 7

Conclusion

Nowadays, mobile video streaming represents the majority of downstream traffic. With large amount of content and increasing demand, the performance of CDNs becomes crucial to ensure user's QoE and content providers' revenue. Our work to enhance the performance of mobile video streaming ecosystems can be summarized as following:

In Chapter 3, we have shown the advantages of applying existing caching mechanisms as a single consistent policy for "BIG" cache, through theoretical analysis and simulation. Our proposed abstraction showed full utilization of the caching resources at intermediate layers, while avoiding the thrashing problem. Moreover, the individual caches have higher hit probability, especially higher layers, as well as the overall cache performance, in terms of minimizing the origin server load and user latency. Also, we introduced *dCLIMB* caching mechanism to minimize the additional overhead of moving objects between boundaries, and showed that it outperforms LRU and K-Hit. Moreover, it is a self-adaptive strategy, which leads to caching popular objects closer to users, and less popular objects closer to origin servers, without the need for prior knowledge of user access patterns, or the need for the usage of timers and counters.

In Chapter 4, we proposed DeepCache Framework, a paradigm which uses the state-of-the-art machine learning tools to improve the performance of content caching. Using such framework, we proposed how to formulate the object characteristics prediction problem as a seq2seq modeling problem. We successfully showed the ability of our LSTM-based model to predict the popularity of content objects. Our results show that enabling DeepCache with existing cache replacement algorithms such as LRU, K-LRU significantly improves their performance. We also discussed how DeepCache framework can be combined with "BIG" cache to be applied for a hierarchical network of cache servers.

In Chapter 5, we presented our proactive resilient routing protocol which aims at ensuring the connectivity between any pair of nodes under arbitrary (link) failures which do not partition the cache network. Our proposed routing algorithm relies on pre-computed routing state and limited local route exchanges or updates based on the key properties of depth-first search. We proved the correctness of our algorithm which ensures the connectivity between any pair of nodes under arbitrary failures without the need for global topology dissemination and route recomputation as in purely distributed routing algorithms or the formation of loops during the convergence process. Our results show that our algorithm limits the number of nodes involved in the recovery process, as well as the number of link reversals, and minimizes the convergence time. An additional advantage is the ability to utilize multiple paths to send traffic between nodes due to utilizing directed edges between nodes even upon failures.

In Chapter 6, our study pointed out both the opportunities and challenges in mmWave 5G, and showed that existing video streaming applications do not work well over mmWave 5G. We proposed new mechanisms "adaptive content bursting" and "dynamic radio switching" to make video streaming applications "5G-aware", and derived theoretical bounds for choosing the appropriate bitrate to avoid stalls during video delivery. We managed to show that with more judicious usage of 5G/4G and appropriate bitrate selection, we can eliminate the stall time while streaming a video with constant bitrate improving the application performance under 5G's challenging network conditions.

We also highlighted at the end of each chapter future research directions to further improve the performance of the mobile video streaming ecosystems.

References

[1] Joan Feigenbaum, Brighten Godfrey, Aurojit Panda, Michael Schapira, Scott Shenker, and Ankit Singla. On the resilience of routing tables. *arXiv preprint arXiv:1207.3732*, 2012.

[2] Sandvine. The Mobile Internet Phenomena Report - February 2020.

[3] Cisco. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, (2017-2022) White Paper.

[4] Vijay Kumar Adhikari, Sourabh Jain, Yingying Chen, and Zhi-Li Zhang. Vivisecting youtube: An active measurement study. In *INFOCOM, 2012.*

[5] Vijay Kumar Adhikari, Yang Guo, Fang Hao, Matteo Varvello, Volker Hilt, Moritz Steiner, and Zhi-Li Zhang. Unreeling netflix: Understanding and improving multi-cdn movie delivery. In *INFOCOM, 2012.*

[6] V. K. Adhikari, Y. Guo, F. Hao, V. Hilt, and Z.L. Zhang. A tale of three CDNs: An active measurement study of Hulu and its CDNs. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*, pages 7–12. IEEE, 2012.

[7] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The akamai network: A platform for high-performance internet applications. *SIGOPS*, 2010.

[8] Cisco. Cisco Visual Networking Index: Forecast and Methodology, 2017-2022.

[9] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking Named Content. In

Proceedings of CoNEXT 2019, CoNEXT '09, pages 1–12, New York, NY, USA, 2009. ACM.

[10] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A Data-oriented (and Beyond) Network Architecture. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 181–192, New York, NY, USA, 2007. ACM.

[11] Eman Ramadan, Arvind Narayanan, and Zhi-Li Zhang. CONIA: Content (provider)-oriented, namespace-independent architecture for multimedia information delivery. In *2015 IEEE International Conference on Multimedia Expo Workshops (ICMEW)*, pages 1–6, June 2015.

[12] Qualcomm. Everything you need to know about 5g. `https://www.qualcomm.com/invention/5g/what-is-5g`. Last Accessed November 2022.

[13] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *CoNEXT*, 2009.

[14] Seyed Kaveh Fayazbakhsh, Yin Lin, Amin Tootoonchian, Ali Ghodsi, Teemu Koponen, Bruce Maggs, K.C. Ng, Vyas Sekar, and Scott Shenker. Less pain, most of the gain: Incrementally deployable icn. In *SIGCOMM, 2013.*

[15] David Starobinski and David Tse. Probabilistic methods for web caching. *Performance evaluation*, 2001.

[16] Vimal Mathew, Ramesh K Sitaraman, and Prashant Shenoy. Energy-aware load balancing in content delivery networks. In *INFOCOM, 2012.*

[17] Alec Wolman, M Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M Levy. On the scale and performance of cooperative web proxy caching. In *SIGOPS*, 1999.

[18] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, 2010.

[19] Valentina Martina, Michele Garetto, and Emilio Leonardi. A unified approach to the performance analysis of caching systems. In *INFOCOM, 2014*.

[20] Asit Dan and Don Towsley. *An approximate analysis of the LRU and FIFO buffer replacement schemes*, volume 18. ACM, 1990.

[21] Hao Che, Zhijung Wang, and Ye Tung. Analysis and design of hierarchical web caching systems. In *INFOCOM, 2001*.

[22] Elisha J Rosensweig, Jim Kurose, and Don Towsley. Approximate models for general cache networks. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[23] Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for lru cache performance. In *Proceedings of the 24th International Teletraffic Congress*, page 8. International Teletraffic Congress, 2012.

[24] N Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. Analysis of ttl-based cache networks. In *Performance Evaluation Methodologies and Tools (VALUETOOLS), 2012 6th International Conference on*, pages 1–10. IEEE, 2012.

[25] P. Babaie, E. Ramadan, and Z. Zhang. Cache network management using big cache abstraction. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 226–234, April 2019.

[26] Weibo Chu, Mostafa Dehghan, Don Towsley, and Zhi-Li Zhang. On allocating cache resources to content providers. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*, ACM-ICN '16, pages 154–159, New York, NY, USA, 2016. ACM.

[27] Andrés Ferragut, Ismael Rodriguez, and Fernando Paganini. Optimizing TTL caches under heavy-tailed demands. In *SIGMETRICS*, 2016.

[28] Zhen Liu, Philippe Nain, Nicolas Niclausse, and Don Towsley. Static caching of web servers. In *Multimedia Computing and Networking 1998*.

[29] Mostafa Dehghan, Laurent Massoulie, Don Towsley, et al. A utility optimization approach to network cache design. In *INFOCOM*. IEEE, 2016.

[30] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017.

[31] A. Sadeghi, F. Sheikholeslami, and G. B. Giannakis. Optimal and scalable caching for 5g using reinforcement learning of space-time popularities. *IEEE Journal of Selected Topics in Signal Processing*, 12(1):180–190, Feb 2018.

[32] Chenyu Li, Jun Liu, and Shuxin Ouyang. Characterizing and predicting the popularity of online videos. *IEEE Access*, 4, 2016.

[33] Wai-Xi andothers Liu. Content popularity prediction and caching for ICN: A deep learning approach with SDN. *IEEE access*, 6, 2018.

[34] Muhammad Zubair Shafiq, Alex X. Liu, and Amir R. Khakpour. Revisiting caching in content delivery networks. volume 42, pages 567–568, New York, NY, USA, June 2014. ACM.

[35] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive ttl-based caching for content delivery. volume 45, pages 45–46, New York, NY, USA, June 2017. ACM.

[36] Wenting Tang, Yun Fu, Ludmila Cherkasova, and Amin Vahdat. Medisyn: A synthetic streaming media service workload generator. In *NOSSDAV*. ACM, 2003.

[37] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[38] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[39] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages II–1764–II–1772. JMLR.org, 2014.

[40] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[41] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[42] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[43] Milad Hashemi et al. Learning memory access patterns. *arXiv preprint arXiv:1803.02329*, 2018.

[44] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. NIPS'14. MIT Press, 2014.

[45] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[46] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[48] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 350–361. ACM, 2011.

[49] G. Iannaccone, C. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an IP backbone. In *Proc. of the 2nd ACM SIGCOMM Workshop on Internet measurment*, page 242. ACM, 2002.

[50] Aman Shaikh, Chris Isett, Albert Greenberg, Matthew Roughan, and Joel Gottlieb. A case study of ospf behavior in a large enterprise network. In *Proceedings*

of the 2nd ACM SIGCOMM Workshop on Internet measurment, pages 217–230. ACM, 2002.

[51] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, and Christophe Diot. Characterization of failures in an ip backbone. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2307–2317. IEEE, 2004.

[52] C. Boutremans, G. Iannaccone, and C. Diot. Impact of link failures on VoIP performance. In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 63–71. ACM New York, NY, USA, 2002.

[53] P. Francois and O. Bonaventure. An evaluation of IP-based fast reroute techniques. In *Proceedings of the 2005 ACM conference on Emerging network experiment and technology (CoNext'05)*, pages 244–245. ACM New York, NY, USA, 2005.

[54] S. Nelakuditi, S. Lee, Y. Yu, and Z.-L. Zhang. Failure insensitive routing for ensuring service availability. In *Proc. IEEE/IFIP IWQoS 2003, Lecture Notes in Computer Science*, pages 287–304, June 2003.

[55] S. Lee, Y. Yu, S. Nelakuditi, Z.-L. Zhang, and C.-N. Chuah. Proactive vs reactive approaches to failure resilient routing. In *Proc. IEEE INFOCOM'04*, March 2004.

[56] Srihari Nelakuditi, Sanghwan Lee, Yinzhe Yu, Zhi-Li Zhang, and Chen-Nee Chuah. Fast local rerouting for handling transient link failures. *IEEE/ACM Transactions on Networking (ToN)*, 15(2):359–372, 2007.

[57] Amund Kvalbein, Audun Fosselie Hansen, Tarik Cicic, Stein Gjessing, and Olav Lysne. Fast ip network recovery using multiple routing configurations. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–11. IEEE, 2006.

[58] Ping Pan, George Swallow, and Alia Atlas. Fast reroute extensions to rsvp-te for lsp tunnels, 2005.

[59] Xiaowei Yang, David Clark, and Arthur W Berger. Nira: a new inter-domain routing architecture. *Networking, IEEE/ACM Transactions on*, 15(4):775–788, 2007.

[60] M. Shand and S. Bryant. IP fast reroute framework. Internet draft, Internet Engineering Task Force, June 2009. (Work in progress).

[61] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[62] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 39–50. ACM, 2009.

[63] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. *SIGCOMM*, 2009.

[64] Ankit Singla, Chi-Yao Hong, Lucian Popa, and Philip Brighten Godfrey. Jellyfish: Networking data centers randomly. In *NSDI*, volume 12, 2012.

[65] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 283–294. ACM, 2013.

[66] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514. ACM, 2014.

[67] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 465–478. ACM, 2015.

[68] Saikat Ray, Roch Guérin, Kin-Wah Kwong, and Rute Sofia. Always acyclic distributed path computation. *IEEE/ACM Transactions on Networking (ToN)*, 18(1):307–319, 2010.

[69] Eli M Gafni and Dimitri P Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communications*, 29(1):11–18, 1981.

[70] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring connectivity via data plane mechanisms. In *NSDI*, pages 113–126, 2013.

[71] Michael Borokhovich, Liron Schiff, and Stefan Schmid. Provable data plane connectivity with local fast failover: Introducing openflow graph algorithms. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 121–126. ACM, 2014.

[72] Jose J Garcia-Lunes-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking (TON)*, 1(1):130–141, 1993.

[73] Pierre Francois and Olivier Bonaventure. Avoiding transient loops during the convergence of link-state routing protocols. *IEEE/ACM Transactions on Networking (TON)*, 15(6):1280–1292, 2007.

[74] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 323–334. ACM, 2012.

[75] Haldane Peterson, Soumya Sen, Jaideep Chandrashekar, Lixin Gao, Roch Guerin, and Zhi-Li Zhang. Message-efficient dissemination for loop-free centralized routing. *ACM SIGCOMM Computer Communication Review*, 38(3):63–74, 2008.

[76] Yang Zhang, Eman Ramadan, Hesham Mekky, and Zhi-Li Zhang. When raft meets SDN: how to elect a leader and reach consensus in an unruly network. In Kai Chen and Jitendra Padhye, editors, *Proceedings of the First Asia-Pacific Workshop on Networking, APNet 2017, Hong Kong, China, August 3-4, 2017*, pages 1–7. ACM, 2017.

[77] Jennifer L Welch and Jennifer E Walter. Link reversal algorithms. *Synthesis Lectures on Distributed Computing Theory*, 2(3):1–103, 2011.

[78] Srihari Nelakuditi, Sanghwan Lee, Yinzhe Yu, Junling Wang, Zifei Zhong, Guor-Huar Lu, and Zhi-Li Zhang. Blacklist-Aided Forwarding in Static Multihop Wireless Networks. In *Proc. of SECON*, Santa Clara, CA, September 2005.

[79] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, and Ion Stoica. Achieving convergence-free routing using failure-carrying packets. *ACM SIGCOMM Computer Communication Review*, 37(4):241–252, 2007.

[80] Surender Baswana, Shreejit Ray Chaudhury, Keerti Choudhary, and Shahbaz Khan. *Dynamic DFS in Undirected Graphs: breaking the O(¡italic¿m¡/italic¿) barrier*, pages 730–739. https://epubs.siam.org/doi/pdf/10.1137/1.9781611974331.ch52.

[81] NS-3 Simulator. `https://www.nsnam.org/`.

[82] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring isp topologies with rocketfuel. *ACM SIGCOMM Computer Communication Review*, 32(4):133–145, 2002.

[83] Cisco. Cisco Visual Networking Index (VNI) Global and Americas/EMEAR Mobile Data Traffic Forecast, (2017–2022).

[84] Verizon's 5G network now available in New York City and more areas. `https://www.androidauthority.com/verizon-5g-916577/`. Last Accessed November 2022.

[85] AT&T Enhances Spectrum Position Following FCC Auction 102. https://about.att.com/story/2019/att_enhances_spectrum_position.html. Last Accessed November 2022.

[86] T-Mobile revs up 5G in 6 cities using mmWave spectrum. https://www.fiercewireless.com/wireless/t-mobile-revs-up-5g-6-cities-using-mmwave-spectrum. Last Accessed November 2022.

[87] 5G has arrived – here's what you can expect from Sprint. https://www.androidauthority.com/what-to-expect-from-sprint-5g-918040/. Last Accessed November 2022.

[88] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Z. Morley Mao, Feng Qian, and Zhi-Li Zhang. A variegated look at 5g in the wild: Performance, power, and qoe implications. *ACM SIGCOMM'21*, 2021.

[89] Arvind Narayanan, Eman Ramadan, et al. Lumos5G: Mapping and Predicting Commercial MmWave 5G Throughput. In *ACM IMC'20*, 2020.

[90] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 272–285, 2016.

[91] Anders Hillbur. 5g deployment options to reduce the complexity. https://www.ericsson.com/en/blog/2018/11/5g-deployment-options-to-reduce-the-complexity. Last Accessed November 2022.

[92] Sylvain Collonge, Gheorghe Zaharia, and G EL Zein. Influence of the human activity on wide-band characteristics of the 60 ghz indoor radio channel. *IEEE Transactions on Wireless Communications*, 3(6):2396–2406, 2004.

[93] Muhammad Kumail Haider, Yasaman Ghasempour, Dimitrios Koutsonikolas, and Edward W Knightly. Listeer: mmwave beam acquisition and steering by tracking

indicator leds on wireless aps. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 273–288. ACM, 2018.

[94] Sanjib Sur, Vignesh Venkateswaran, Xinyu Zhang, and Parmesh Ramanathan. 60 GHz indoor networking through flexible beams: A link-level profiling. In *ACM SIGMETRICS Performance Evaluation Review*, volume 43, pages 71–84. ACM, 2015.

[95] Hang Zhao, Rimma Mayzus, Shu Sun, Mathew Samimi, Jocelyn K. Schulz, Yaniv Azar, Kevin Wang, George N. Wong, Felix Gutierrez, and Theodore S. Rappaport. 28 GHz millimeter wave cellular communication measurements for reflection and penetration loss in and around buildings in New York city. In *2013 IEEE International Conference on Communications, ICC 2013*, pages 5163–5167, 2013.

[96] Theodore S Rappaport, Felix Gutierrez, Eshar Ben-Dor, James N Murdock, Yijun Qiao, and Jonathan I Tamir. Broadband millimeter-wave propagation measurements and models using adaptive-beam antennas for outdoor urban cellular communications. *IEEE transactions on antennas and propagation*, 61(4):1850–1859, 2013.

[97] Theodore S Rappaport, Shu Sun, Rimma Mayzus, Hang Zhao, Yaniv Azar, Kevin Wang, George N Wong, Jocelyn K Schulz, Mathew Samimi, and Felix Gutierrez. Millimeter wave mobile communications for 5G cellular: It will work! *IEEE access*, 1:335–349, 2013.

[98] Marco Giordani, Marco Mezzavilla, and Michele Zorzi. Initial Access in 5G mmWave Cellular Networks. *IEEE Communications Magazine*, 54(11):40–47, 2016.

[99] Joan Palacios, Danilo De Donno, and Joerg Widmer. Tracking mm-Wave channel dynamics: Fast beam training strategies under mobility. In *Proceedings of the IEEE Conference on Computer Communications*, 2017.

[100] Wonil Roh, Ji-Yun Seol, Jeongho Park, Byunghwan Lee, Jaekon Lee, Yungsoo Kim, Jaeweon Cho, Kyungwhoon Cheun, and Farshid Aryanfar. Millimeter-wave

beamforming as an enabling technology for 5G cellular communications: theoretical feasibility and prototype results. *IEEE Communications Magazine*, 52(2):106–113, 2014.

[101] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. A first look at commercial 5g performance on smartphones. In *Proceedings of The Web Conference 2020*, WWW '20, page 894–905, New York, NY, USA, 2020. Association for Computing Machinery.

[102] Xuan Kelvin Zou, Jeffrey Erman, Vijay Gopalakrishnan, Emir Halepovic, Rittwik Jana, Xin Jin, Jennifer Rexford, and Rakesh K. Sinha. Can accurate predictions improve video streaming in cellular networks? In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, HotMobile '15, page 57–62, New York, NY, USA, 2015. Association for Computing Machinery.

[103] Aaron Schulman, Vishnu Navda, Ramachandran Ramjee, Neil Spring, Pralhad Deshpande, Calvin Grunewald, Kamal Jain, and Venkata N Padmanabhan. Bartendr: a practical approach to energy-aware cellular data scheduling. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, pages 85–96. ACM, 2010.

[104] Robert Margolies, Ashwin Sridharan, Vaneet Aggarwal, Rittwik Jana, NK Shankaranarayanan, Vinay A Vaishampayan, and Gil Zussman. Exploiting mobility in proportional fair cellular scheduling: Measurements and algorithms. *IEEE/ACM Transactions on Networking (TON)*, 24(1):355–367, 2016.

[105] Mohammad Hosseini and Christian Timmerer. Dynamic adaptive point cloud streaming. In *Proceedings of the 23rd Packet Video Workshop*, PV, 2018.

[106] Feng Qian, Bo Han, et al. Toward practical volumetric video streaming on commodity smartphones. In *HotMobile*, 2019.

[107] Bo Han, Yu Liu, and Feng Qian. Vivo: Visibility-aware mobile volumetric video streaming. In *ACM MobiCom*, 2020.

[108] Susanna Schwarzmann, Clarissa Cassales Marquezan, et al. Estimating Video Streaming QoE in the 5G Architecture Using Machine Learning. In *Internet-QoE*, 2019.

[109] J. Qiao, Y. He, and X. S. Shen. Proactive caching for mobile video streaming in millimeter wave 5g networks. *IEEE Transactions on Wireless Communications*, 15(10):7187–7198, 2016.

[110] CellInfoNr — Android Developers. `https://developer.android.com/reference/kotlin/android/telephony/CellInfoNr`. Last Accessed November 2022.

[111] CellIdentityNr — Android Developers. `https://developer.android.com/reference/kotlin/android/telephony/CellIdentityNr`. Last Accessed November 2022.

[112] CellSignalStrengthNr — Android Developers. `https://developer.android.com/reference/kotlin/android/telephony/CellSignalStrengthNr`. Last Accessed November 2022.

[113] Verizon Coverage Maps. `https://www.verizon.com/5g/coverage-map/?city=minneapolis`. Last Accessed November 2022.

[114] iPerf3 – iPerf 3.7 documentation. `https://software.es.net/iperf/`. Last Accessed November 2022.

[115] Speedtest by ookla. `https://www.speedtest.net/`. Last Accessed November 2022.

[116] Qualcomm. Mobilizing mmwave for smartphones. `https://www.qualcomm.com/news/onq/2019/02/13/track-solve-another-impossible-challenge-mobilizing-mmwave-smartphones`. Last Accessed November 2022.

[117] M. Belshe, R. Peon, and Ed. M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, Internet Engineering Task Force, 2015.

[118] T. Golla and R. Klein. Real-time point cloud compression. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5087–5092, Sep. 2015.

[119] Yan Huang, Jingliang Peng, C. C. Jay Kuo, and M. Gopi. A generic scheme for progressive point cloud coding. *IEEE Transactions on Visualization and Computer Graphics*, 14(2):440–453, March 2008.

[120] Ruwen Schnabel and Reinhard Klein. Octree-based point-cloud compression. In *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG'06, page 111–121, Goslar, DEU, 2006. Eurographics Association.

[121] Feng Qian, Bo Han, Jarrell Pair, and Vijay Gopalakrishnan. Toward practical volumetric video streaming on commodity smartphones. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 135–140. ACM, 2019.

[122] Qi He, Constantine Dovrolis, and Mostafa Ammar. On the predictability of large transfer tcp throughput. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 145–156. ACM, 2005.

[123] Traffic control in the linux kernel. `https://linux.die.net/man/8/tc/`. Last Accessed November 2022.

[124] H.264/MPEG-4 AVC. `http://handle.itu.int/11.1002/1000/6312`. Last Accessed November 2022.

[125] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the H. 264/AVC standard. *IEEE Transactions on circuits and systems for video technology*, 17(9):1103–1120, 2007.

[126] The 5g status quo is clearly not good enough. `https://www.t-mobile.com/news/the-5g-status-quo-is-clearly-not-good-enough`. Last Accessed November 2022.

[127] 5g spectrum: strategies to maximize all bands. `https://www.ericsson.com/en/networks/trending/hot-topics/`

`5g-spectrum-strategies-to-maximize-all-bands`. Last Accessed November 2022.

[128] 5g low latency requirements. `https://broadbandlibrary.com/5g-low-latency-requirements/`. Last Accessed November 2022.

[129] Snapdragon x55 5g modem-rf system. `https://www.qualcomm.com/products/snapdragon-x55-5g-modem`. Last Accessed November 2022.

[130] Zhi-Li Zhang, Udhaya K. Dayalan, Eman Ramadan, and Timothy J. Salo. Towards a software-defined, fine-grained qos framework for 5g and beyond networks. In *Proceedings of the ACM SIGCOMM Workshop on Network Meets AI & ML*, NetAI'21, 2021.

Appendix A

Publications

In addition to this dissertation, the presented work and results are also documented in the following published papers.

A.1 Publications by Date

Under Submission:

- Eman Ramadan, Tu Nguyen, Zhi-Li Zhang. Loop-free Resilient Routing Under Arbitrary Network Failures. Under Submission, 2023.

2023:

- Rostand A. K. Fezeu , Eman Ramadan, Wei Ye, Benjamin Minneci, Jack Xie, Arvind Narayanan, Ahmad Hassan, Feng Qian, Zhi-Li Zhang, Jaideep Chandrashekar, Myungjin Lee. An In-Depth Measurement Analysis of 5G mmWave PHY Latency and its Impact on End-to-End Delay. In the Passive and Active Measurement Conference (PAM), 2023.

2022:

- Xinyue Hu, Eman Ramadan, Wei Ye, Feng Tian, Zhi-Li Zhang. Raven: Belady-Guided, Predictive (Deep) Learning for In-Memory and Content Caching. In the Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2022.

2021:

- Eman Ramadan, Hesham Mekky, Cheng Jin, Braulio Dumba, Zhi-Li Zhang. Taproot: Resilient Diversity Routing with Bounded Latency. In the Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR), 2021.

- Eman Ramadan, Arvind Narayanan, Udhaya Kumar Dayalan, Rostand A. K. Fezeu, Feng Qian, Zhi-Li Zhang. Case for 5G-aware Video Streaming Applications. In the Proceedings of the 1st Workshop on 5G Measurements, Modeling, and Use Cases (5G-MeMU), 2021.

- Zhi-Li Zhang, Udhaya Kumar Dayalan, Eman Ramadan, Timothy J. Salo. Towards a Software-Defined, Fine-Grained QoS Framework for 5G and Beyond Networks. In the Proceedings of the ACM SIGCOMM 2021 Workshop on Network-Application Integration (NAI), 2021.

2020:

- Arvind Narayanan, Eman Ramadan, Rishabh Mehta, Xinyue Hu, Qingxu Liu, Rostand A. K. Fezeu, Udhaya Kumar Dayalan, Saurabh Verma, Peiqi Ji, Tao Li, Feng Qian, Zhi-Li Zhang. Lumos5G: Mapping and Predicting Commercial mmWave 5G Throughput. In ACM Internet Measurement Conference (IMC), 2020.

- Arvind Narayanan, Eman Ramadan, Jacob Quant, Peiqi Ji, Feng Qian, Zhi-Li Zhang. 5G Tracker - A Crowdsourced Platform to Enable Research Using Commercial 5G Services. ACM SIGCOMM Posters, 2020.

- Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. A First Measurement Study of Commercial mmWave 5G Performance on Smartphones. In The Web Conference (WWW), 2020.

2019:

- Eman Ramadan, Pariya Babaie, Zhi-Li Zhang. Performance Estimation and Evaluation Framework for Caching Policies in Hierarchical Caches. In Computer Communications, Volume 144. Computer Communications, 2019.

- Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, Zhi-Li Zhang. Making Content Caching Policies 'Smart' Using the DEEPCACHE Framework. In ACM SIGCOMM Computer Communication Review (SIGCOMM CCR), 2019.
- Pariya Babaie, Eman Ramadan, Zhi-Li Zhang. Cache Network Management Using BIG Cache Abstraction. In IEEE Conference on Computer Communications (INFOCOM), 2019.

2018:

- Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, Zhi-Li Zhang. DeepCache: A Deep Learning Based Framework For Content Caching. In Workshop on Network Meets AI & ML, (SIGCOMM WKSHPS NetAI), 2018.
- Eman Ramadan, Pariya Babaie, Zhi-Li Zhang. A Framework for Evaluating Caching Policies in a Hierarchical Network of Caches. In IFIP Networking Conference and Workshops, (IFIP Networking), 2018.
- Arvind Narayanan, Eman Ramadan, Zhi-Li Zhang. OpenCDN: An ICN-based Open Content Distribution System Using Distributed Actor Model. In IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS IECCO), 2018.

2017:

- Yang Zhang, Eman Ramadan, Hesham Mekky, Zhi-Li Zhang. When Raft Meets SDN: How to Elect a Leader and Reach Consensus in an Unruly Network. In Asia-Pacific Workshop on Networking, (APNet) 2017.
- Eman Ramadan, Arvind Narayanan, Zhi-Li Zhang, Runhui Li, Gong Zhang. BIG Cache Abstraction for Cache Networks. In The 37th IEEE International Conference on Distributed Computing Systems, (ICDCS) 2017.

2016:

- Eman Ramadan, Hesham Mekky, Braulio Dumba, Zhi-Li Zhang. Adaptive Resilient Routing via Preorders in SDN. In Workshop on Distributed Cloud Computing, (DCC) 2016.

2015:

- Eman Ramadan, Arvind Narayanan, Zhi-Li Zhang. CONIA: Content (provider)-Oriented, Namespace-Independent Architecture for Multimedia Information Delivery. In Workshop on Multimedia & Expo. (ICMEW), 2015.