

© 2026 William Sentosa

Low-Latency Network Transport Services Over Heterogeneous Paths

by

William Sentosa

Dissertation

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2026

Urbana, Illinois

Doctoral Committee:

Professor Philip Brighten Godfrey, Chair
Professor Sarita V. Adve
Assistant Professor Elahé Soltanaghahi
Professor Bruce MacDowell Maggs

Abstract

Interactive mobile applications such as web browsing and extended reality (XR) benefit significantly from low-latency networking, as they frequently communicate with cloud servers and other users' devices. At the same time, they require moderate to high bandwidth to support rich media and real-time interaction. Achieving both low latency and high bandwidth within a single access network, however, is fundamentally difficult due to the inherent trade-off between the two, particularly in wireless environments.

To address this trade-off, emerging network technologies such as 5G New Radio (NR) and Wi-Fi are beginning to provide heterogeneous paths (or channels), each optimized for one dimension. For example, in 5G NR, URLLC (a low-latency channel or LLC) offers consistent low latency but limited bandwidth, while eMBB (a high-bandwidth channel or HBC) provides higher throughput with more variable latency. This development creates opportunities for applications to exploit multiple paths jointly to achieve better performance than any single path alone.

This dissertation presents and validates the hypothesis that applications can achieve significant performance gains by utilizing HBC and LLC in parallel. It contributes to the design of low-latency transport services across networking layers over heterogeneous paths, as well as to the development of an accurate evaluation tool for new networking protocols. The first contribution is DChannel, a network-layer system that dynamically steers packets across HBC and LLC using a cost-reward model, improving web and mobile application responsiveness by 16–40% in real 5G environments without requiring application modifications. The second contribution is DChannelX, an application-transport layer solution built on MPQUIC that also dynamically steers traffic across HBC and LLC and leverages application-level knowledge such as message boundaries and priorities to further optimize channel utilization. We show that

DChannelX achieves better performance than DChannel. The third contribution is CellReplay, which establishes the need for accurate evaluation methodologies and presents a record-and-replay network emulation system to support networking protocol design evaluation under dynamic cellular network conditions. Evaluations show that applications (e.g., web browsing and video streaming) running on CellReplay closely match real-world performance with significantly lower error than prior approaches.

Acknowledgements

This PhD journey has been an enjoyable, challenging, and truly rewarding experience, one that I will probably only go through once in a lifetime. There are so many people I want to thank: my advisor, collaborators, friends, and family. I would also like to thank my committee members: Prof. Sarita Adve, Prof. Elahe Soltanaghahi, and Prof. Bruce Maggs, for their valuable feedback which helped shape this thesis. None of this would have been possible without their support.

First of all, I am very grateful to my amazing advisor, Prof. Brighten Godfrey, for teaching me how to do networking research and identify interesting problems to solve. You helped me grow into a better and stronger person. I have been incredibly lucky to work with you – from my second year in the PhD program all the way to graduation – especially after my rocky first year. Throughout those years, I have enjoyed our discussions on research, life, and many other things. From you, I learned how to write papers, pitch ideas, make presentations, and collaborate with people. You always believed in me when I did not have confidence in myself. You have also given me the freedom to explore what I wanted to do and supported me every step of the way.

I have also been fortunate to be mentored by some of the best researchers throughout grad school. Early on, I worked on the Speed of Light Internet project with Prof. Bruce Maggs (Duke University), Prof. Bala Chandrasekaran (VU Amsterdam), and Prof. Ankit Singla (ETH Zurich), which eventually led to DChannel. I am also very thankful to Prof. Haitham Hassanieh (EPFL), one of my closest collaborators, for his expertise in cellular networks, sharp technical insights, and great paper-writing advice. We have published DChannel and CellReplay together. I also want to thank my other collaborators as well, such as Debopam and Waqar.

I want to thank Prof. Sarita Adve and her team (especially Qinjun) for our collaboration on XR offloading projects, where I got to apply my low-latency ideas

directly to XR applications through ILLIXR. I was glad to see that ILLIXR uses and benefits from my cellular network emulator. I was also glad to contribute to the open-source ILLIXR codebase and be a part of that community. Our collaboration even led to two papers at ACM MMSys.

I want to thank Prof. Haryadi Gunawi (UChicago) for opening research opportunities for Indonesian undergraduates and connecting me with Prof. Shan Lu, who first introduced me to systems research and became an incredible mentor. I was fortunate to co-author an ASPLOS paper with her, and she was also the one who encouraged me to apply to the UIUC CS PhD program, where she is an alumna. Her mentorship and support are a big reason why I am here in the first place.

Thanks also go to all members of the networking research groups at UIUC, especially those from Brighten's and Radhika's groups. I really enjoyed our group meetings, social gatherings, and all the fun times we had together. Big thanks to Sangeetha, Nathan, Tong, and Vipul for helping me during my early years. Thanks to Milind for all the countless dinners and great conversations we had. I am also grateful to my lab mates from Brighten's group — Sachin, Talal, Shehab, Gangmuk, Rahul, Gerard, Talha, and Wanyu — and from other groups — Ammar, Yongzhou, Taimoor, Eashan, Ayush, and Emerson. I also want to thank everyone from Sarita's group. I really enjoyed our interactions, and thank you for inviting me to your group dinners. It was always a wonderful time.

I want to thank all of my friends. I do not think I could have survived the PhD journey without their support. A special thanks to my first roommates in the United States, Alif and Owen, who made me feel welcome when I first moved to the US and UIUC. You both cheered me up during some of the toughest times in my first year. I want to thank Alvin, who was always up for lunch or dinner whenever I needed a break on stressful days, and Felix, my second roommate, for being a great roommate. Thanks also to Fery, Aldi, Tio, and Matt for inviting me to travel around the US. It was a

much-needed refreshment for me. A special big thanks to Titan, who kindly let me crash at his apartment countless times when I needed to go to Chicago to run experiments for my research. I also want to thank Jennifer for all the lunches and dinners we had around campus, and for all the PhD life talks.

Finally, I want to thank my family for their love and support. Of course, I would not be here without support from them.

Contents

Chapter 1	Introduction	10
1.1	Motivation	10
1.2	Challenges and Opportunities	11
1.3	Contribution	13
1.4	Dissertation-related Publications	16
Chapter 2	Background	17
2.1	Variety of the high bandwidth and low latency channels	17
2.2	Multipath networking and related work	20
Chapter 3	Accelerating general mobile applications through a network layer steering with DChannel	23
3.1	Background on web browsing traffic	25
3.2	DChannel Design	26
3.3	Prototype and Experimental Setup	35
3.4	Evaluation	41
3.5	Discussions on DChannel for 5G networks and beyond	56
Chapter 4	Informed application-transport service with DChannelX	59
4.1	The need to operate at the transport layer	59
4.2	DChannelX as the application-transport solution	62
4.3	DChannelX implementation	67
4.4	Preliminary evaluation	68
4.5	Discussion on DChannelX and future work	74
Chapter 5	Towards an accurate record-and-replay for cellular network emulation with CellReplay	76
5.1	Background and related work	80
5.2	Live record-and-replay is hard	83
5.3	CellReplay design and implementation	89
5.4	Evaluation	99
5.5	Discussion on CellReplay and future work	115
Chapter 6	Discussions and future work	118
6.1	DChannel and DChannelX in today's 5G deployment	118
6.2	Specialized low latency paths	119

6.3 Beyond cellular network emulation	120
References	122

Chapter 1 Introduction

1.1 Motivation

With the rapid rise of interactive network applications, low latency provided by the network has become critical for delivering a seamless user experience. Web browsing, one of the most popular networked applications, is particularly sensitive to network latency, as it requires making multiple round-trips to servers to retrieve content. Prior studies also show that network round-trip time is a dominant factor that affects the performance of such applications [1], [2]. From an economic perspective, even small increases in latency can have significant consequences. For example, Amazon reported that an additional 100 ms of latency can lead to up to a 1% loss in revenue [3]. Similarly, Akamai found that a 100 ms delay in website load time can hurt conversion rates by 7% [4].

Low latency is even more critical for emerging interactive applications, whose viability depends directly on it. Distributed extended reality (XR), including virtual (VR) and augmented reality (AR), enables distant collaboration in a shared 3D environment, but this is only possible when network latency is minimal. There is also a growing shift toward cloud-based XR, where computationally intensive processes are offloaded to edge servers to reduce power consumption and accommodate lightweight devices [5], [6]. This approach further increases the demand for low and consistent latency. Prior studies [7], [8], [9] show that VR systems require motion-to-photon latency below 20 ms to prevent simulator sickness, whereas AR systems impose an even stricter requirement, typically below 5 ms.

In addition to low latency, these applications also require networks with moderate to high bandwidth. A study conducted in 2013 showed that web page loads require approximately 16 Mbps of bandwidth for the bottleneck to shift from bandwidth to network latency. As web pages have become increasingly complex and larger over time, with more objects to fetch [10], modern web browsing requires a moderate amount of

bandwidth, particularly to handle bursts of requests. XR applications are also bandwidth-intensive. They require high throughput to stream high-quality rendered content [6] or to transmit camera frames for tasks such as pose estimation [5]. For instance, VR requires 190 Mbps of network bandwidth to deliver rendered frames [6], based on the resolution and refresh rate of devices such as the Valve Index [11].

1.2 Challenges and Opportunities

Despite the demanding low-latency requirements of modern networked applications, currently deployed 5G mobile networks still fall short of meeting these demands. According to our measurements (§3.3.2), commercial 5G (Verizon 5G mmWave) can achieve a median RTT of 22 ms under ideal conditions (e.g., stationary scenarios with minimal contention). However, even in such ideal settings, tail latency remains high. For instance, our measurements show that P98 latency can spike to as high as 106 ms under these conditions.

We also find that latency further degrades in more challenging environments, such as during mobility. Figure 1 shows the end-to-end RTT of 1400-byte packets sent back-to-back every 15 ms to an echo server while driving in downtown Chicago over Verizon 5G mmWave, with RTT frequently spiking to 100–300 ms. In conclusion, these results show that providing consistently low-latency service in mobile networks is still challenging.

In response, network providers are beginning to explore specialized services that aim to provide consistently low latency. 5G New Radio (NR) supports ultra-low end-to-end latency (2–10 ms) with high reliability (99.999%) through URLLC. However, this low latency comes at a significant cost in bandwidth, as URLLC typically provides only 0.4–16 Mbps [12], making it unsuitable for many popular applications. Consequently, URLLC primarily targets niche, mission-critical use cases such as remote surgery, remote driving, and factory automation. Most user-facing applications (e.g., web browsing and

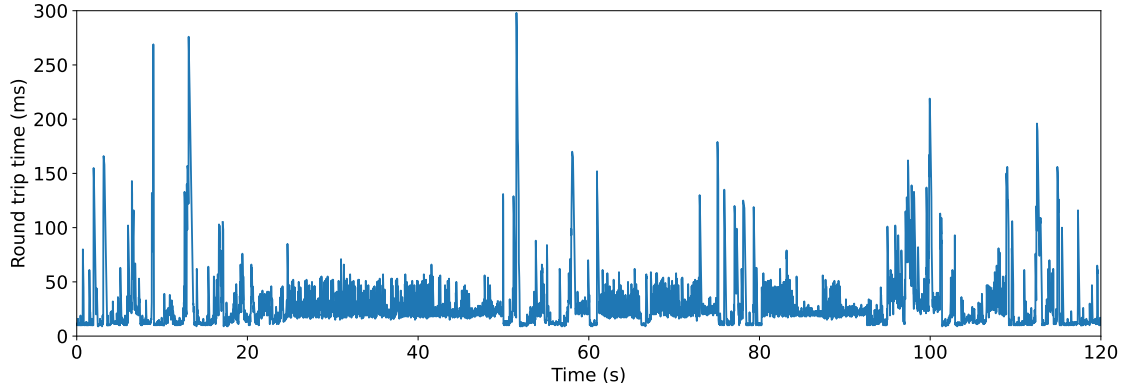


Figure 1: End-to-end packet round-trip-time measured while driving in downtown Chicago with Verizon 5G mmWave.

video streaming) rely on 5G eMBB, which prioritizes high throughput with less emphasis on low latency. eMBB is the service commonly provided by network operators (such as Verizon) for their regular customers and corresponds to the network conditions observed in our measurements. This separation of services exists because of a fundamental limitation: achieving both low latency and high bandwidth within a single access network is inherently difficult due to the well-known latency–throughput trade-off, particularly in wireless environments [13].

Beyond mobile networks, we also see that paths with latency and bandwidth heterogeneity and trade-offs extend to other network types. For instance, Wi-Fi also introduces a similar deterministic low-latency channel, wireless time-sensitive networking (WTSN) [14], which can coexist with its regular channel that provides high bandwidth [15]. Wide area network (WAN) paths are also diversifying such that paths may have a latency-bandwidth trade-off. An example is Low-Earth Orbit satellite networks (e.g., SpaceX Starlink[16]), which could offer much lower latency than the terrestrial Internet over longer distances but with lower bandwidth.

Given the availability of these heterogeneous paths (or channels), the common way to leverage the high-bandwidth channel (HBC) and the low-latency channel (LLC) is to allow application traffic to be statically mapped to a particular channel according to its

specific needs. For example, in 5G, moderate- to high-bandwidth applications such as general mobile applications (e.g., media streaming and web browsing) will use eMBB, while mission-critical applications (e.g., remote surgery and factory automation) that only send small amounts of traffic will use URLLC. We believe, however, that the availability of two channels offers an opportunity to deal with the fundamental latency–bandwidth trade-off in a *new* way, beyond simple static assignment of an application to a single channel. Specifically, we present the hypothesis that *by using high-bandwidth (HBC) and low-latency (LLC) channels in parallel, applications can achieve significant improvements in performance and user experience.*

1.3 Contribution

In this dissertation, we *explore directions to utilize these channels across different networking layers to enable interactive applications that require both high throughput and low latency in modern wireless networks.* In addition, we also *design and implement an experimental platform to enable accurate application protocol evaluation under cellular networks.*

The first part of the dissertation (Chapter 3) will describe how to utilize such channels by designing and implementing DChannel, a network-layer approach that steers packets without any application input and modification. DChannel comprises two modules running at either end of the channels – namely, in the mobile device OS and in a gateway device operated by the service provider. Each module runs a packet steering scheme to steer packets (i.e., IP packets) between LLC or HBC. To decide which packets are worth accelerating, since LLC bandwidth is extremely limited, DChannel treats the channel as an *expensive* resource and calculates the estimated benefit and cost of utilizing LLC for each packet. Finally, since the parallel channels could occasionally confuse the transport layer with out-of-order delivery, DChannel employs a reordering buffer in the mobile device and gateway. With 5G channels, our trace-driven and live network

experiments show that even though URLLC offers just 1% of the bandwidth of eMBB, using both channels can improve web page load time and responsiveness of common mobile apps by 16-40% compared to using exclusively eMBB.

In the second part (Chapter 4), we propose DChannelX, an application-transport solution that incorporates application knowledge to better utilize HBC and LLC. A network-layer solution like DChannel may perform sub-optimally because of these two reasons. First, it lacks application information (e.g., flow priority or message boundary) that can be useful information to schedule packets to the channels. Second, it may interfere with application logic (e.g., delay-dependent congestion control like BBR [17]) which may lead to performance degradation, since network-layer steering is transparent to applications. Thus, we developed DChannelX that can leverage application information such as message boundaries and priority to steer packets more effectively across the two paths on top of MPQUIC. Our preliminary experiments show that by leveraging message boundary and priority information in DChannelX, it can achieve better performance compared to DChannel by 15.36%.

The third part (Chapter 5) describes an accurate network emulation system to support application protocol design evaluation under a variety of cellular network conditions. The inherent variability of real-world cellular networks makes it hard to evaluate, reproduce, and debug the performance of networked applications (such as DChannel or DChannelX). Although the gold standard is testing on live networks, such testing is time-consuming, highly variable, and difficult to reproduce. Thus, researchers and developers often rely on emulation for much of their evaluation, hoping to replicate a representative environment that yields performance similar to a live network. A common approach is to *record* network performance traces (e.g., bandwidth) over time using predefined workloads (e.g., Saturator [18] or iPerf) on real networks and then *replay* those traces in an emulated network for other workloads. However, we show that this approach produces inaccurate results and can even introduce evaluation bias. For

example, we observe an average bias of approximately 17.1% in web page load times (PLTs) compared to live network conditions, and network-adaptive applications (e.g., ABR) can behave differently and can overestimate performance improvements.

Chapter 5 investigates the root causes of the inaccuracy of the record-and-replay approach and presents CellReplay, a tool that records the time-varying performance of a live cellular network into traces using preset workloads and faithfully replays the observed performance for other workloads through an emulated network interface. A key challenge is capturing the workload-dependent nature of cellular networks, where available bandwidth depends significantly on the traffic pattern. CellReplay addresses this by recording network behavior under two predefined workloads simultaneously and interpolating during replay for other workloads. Across diverse network conditions, our evaluation shows that real-world applications (e.g., web browsing or video streaming) running on CellReplay achieve performance (e.g., page load time or bitrate selection) similar to their live network counterparts, with significantly reduced error compared to the prior method.

Finally, in Chapter 6, we discuss broader post-thesis research opportunities, including the deployment of DChannel and DChannelX in current 5G networks, how future cellular networks can benefit from the findings of this dissertation, how to create low-latency paths through service differentiation, and research directions beyond cellular network emulation.

In conclusion, this dissertation validates the hypothesis through these three cohesive systems. First, DChannel shows that heterogeneous channels can be effectively utilized at the network layer to improve application performance. Second, DChannelX demonstrates that incorporating application- and transport-layer knowledge further increases these gains, and it highlights the importance of cross-layer design. Third, CellReplay establishes the need for accurate evaluation methodologies when designing new network protocols over cellular networks and introduces a system that enables such evaluation.

1.4 Dissertation-related Publications

DChannel [19] was published at USENIX NSDI'23 and its early version [20] was published at ACM HotMobile'21. A position paper [21] related to the DChannelX was published at ACM HotNets'23, and the corresponding full paper is currently under preparation. CellReplay [22] was published at USENIX NSDI'25, where it received the Community Award for the best paper with open-sourced code or datasets.

Chapter 2 Background

2.1 Variety of the high bandwidth and low latency channels

To support the demanding application requirements, networks have started to provide different types of channels (or services) either supporting high bandwidth or low latency. In this section, we discuss such channels in the context of wireless networks (5G NR and Wi-Fi) and wide area network (WAN) designs.

2.1.1 5G NR

5G wireless networks are designed to support applications with very different service level requirements. The 5G standard known as New Radio (NR) specifies three service models: (1) enhanced mobile broadband (eMBB) for standard high-data-rate Internet and mobile connectivity, (2) ultra-reliable low-latency communication (URLLC) for mission-critical and latency-sensitive applications, and (3) massive machine-type communications (mMTC) for large-scale IoT deployments. We describe eMBB and URLLC in more depth.

Enhanced Mobile Broadband. This service focuses on providing high-data-rate mobile access. It is considered an upgrade to 4G mobile broadband that will satisfy the ever-increasing demand for mobile and wireless data. 5G eMBB can operate either at the low-frequency bands below 6 GHz which we refer to as low-band or the high-frequency bands around 28 GHz/39 GHz which we refer to as millimeter wave (mmWave). The mmWave bands are a key new technology in 5G as they offer $10\times$ the bandwidth that is currently available to 4G LTE networks [23], enabling user throughput of around 1 Gbps [24].

Providers like Verizon, AT&T, and T-Mobile have already deployed both the low-band and mmWave 5G in several major US cities, including Chicago, Atlanta, New York, and Los Angeles [25], [26], [27], [28]. A recent measurement study on commercial

mmWave 5G networks in the US shows TCP throughput of up to 2 Gbps for download and 60 Mbps for upload, with a mean RTT of 28 ms measured between the client and the first-hop edge server right outside the cellular network core [25]. The measurements were performed, however, in conditions favorable to mmWave such as line-of-sight, no mobility, and few clients.

eMBB latency is expected to be higher as the number of users increases and as users move. This is because radio access networks (RANs) operating in the mmWave bands use very directional beams to compensate for high signal attenuation, making them vulnerable to blockage and mobility. High data rate communication is possible only when the RAN access point aligns its beam towards the user [29], [30], [31]. This process, commonly referred to as beam alignment, can introduce significant delays, especially when users are moving, which requires the access point to keep realigning the beam of each user [29], [32]. Furthermore, the user or other obstacles can easily block the beam, leading to unreliable and inconsistent performance both in terms of changes in throughput and highly variable RTT [25], [33], [34]. Our own experiments in Chicago also confirm this and show that the RTT can vary significantly even for stationary clients and is further exacerbated while walking or driving. This is because 5G eMBB mainly optimizes for high data rates, focusing less on reliability and low latency.

Ultra-Reliable Low-Latency Communication. Unlike eMBB, this channel focuses on providing highly reliable, very low latency communication at the cost of limited throughput. It aims to support mission-critical and emerging applications with stringent latency and reliability requirements such as self-driving cars, factory automation, and remote surgery. While the URLLC channel is yet to be deployed in practice, the standard specifies a target 0.5 ms air latency between the client and the RAN (1 ms RTT) with 99.999% reliability for small packets (e.g. 32 to 250 bytes) [24]. It also specifies a target end-to-end latency (from a client to a destination typically right outside

the cellular network core) of 2 to 10 ms with throughput ranging between 0.4 to 16 Mbps depending on the underlying application [35]. URLLC is expected to operate in the sub-6 GHz frequency bands (e.g. 700 MHz or 4 GHz) and operators are expected to use network slicing to provide dedicated resources to URLLC clients in order to guarantee consistent performance in terms of latency and reliability across both the radio access network (RAN) and the cellular core [35]. Finally, client access to the URLLC channel will be controlled by the network operators. The access control network slicing mechanisms, however, are left to the operators' own implementations [12].

2.1.2 Wi-Fi

Emerging applications such as smart factories, collaborative robots, and extended Reality (XR) require Wi-Fi to deliver data in lower latency and higher reliability. In response to these demands, Wi-Fi has started to adopt a Time-Sensitive Networking (TSN) to facilitate data delivery with bounded latency and zero congestion loss.

This is achieved by applying 802.1Qbv (time-aware scheduling) to Wi-Fi as it will reserve resources and protect time-critical traffic from congestion induced latency [36]. Moreover, more aggressive redundancy is also applied to reduce unnecessary latency due to retransmission. In consequence, the Wi-Fi TSN provides low latency but also comes with lower bandwidth compared to the traffic in the regular channel which is unprotected against congestion.

2.1.3 Wide area networks

Wide area networks (WANs) also have started to provide multiple paths with different latency and bandwidth trade-off. Low-Earth Orbit satellite networks, like SpaceX Starlink [16], could offer significantly lower latencies than the terrestrial Internet by leveraging space-based inter-satellite lasers [37], [38] operating at the speed-of-light, albeit with lower bandwidth than fiber due to radio up/downlink bottleneck. These

networks, in conjunction with the terrestrial Internet, can be used as a pair of high bandwidth and low latency channels. Similar pair may also materialize at content delivery networks (CDN) and content provider infrastructures as some of the more novel network architectures (e.g., SCION [39] and cISP [2]) are widely deployed. CDN servers may gain access, for instance, to a microwave-based ultra-low-latency network in addition to the conventional terrestrial fiber optic network [2]. While cISP’s microwave links operating at the speed of light are much faster than optical fiber, their bandwidth and reliability are vastly smaller than those of fiber. Similarly, SCION could help a host learn of the existence of multiple paths to a destination along with the performance characteristics of each such path [39]. SCION effectively transforms the multiple paths (with vastly different characteristics) a host has to a destination into the low latency and high bandwidth paths.

2.2 Multipath networking and related work

The concept of using multiple access networks in parallel is known as multipath networking. Numerous studies have explored this topic across different layers of the network stack, from the network layer to the application layer. Most of these efforts differ from our work in that they consider networks where all paths provide substantial bandwidth and lack explicit bandwidth-latency trade-offs.

2.2.1 Network layer multipath

Network-layer multipath operates by steering IP packets across multiple network paths. This approach is transparent to upper layers and does not require modifications or input from the transport or application layers. DChannel belongs to this category.

Tsao and Sivakumar [40] proposed a super-aggregation concept to improve TCP throughput over Wi-Fi by using a 3G network in parallel with Wi-Fi and selectively steering small packets (e.g., ACKs) to 3G. The key insight is that small ACK packets

contend for transmission equally with data frames at the Wi-Fi MAC layer. By offloading these ACK packets to another path (i.e., 3G), the Wi-Fi MAC resources previously occupied by small ACK packets can instead be used to transmit larger data frames, thereby improving overall Wi-Fi throughput.

ASAP [41] steers packets across a high-bandwidth high-latency satellite ISP and a low-bandwidth low-latency terrestrial network to improve HTTPS performance. This work is similar to ours in that it addresses networks with bandwidth–latency trade-offs. ASAP employs a phase-based offloading strategy that accelerates packets belonging to latency-sensitive phases of HTTPS, such as TCP and TLS/SSL connection setup. In our evaluation, we compared DChannel’s performance against ASAP and found that DChannel performs better when using high-bandwidth and low-latency channels, as it makes finer-grained scheduling decisions.

2.2.2 Transport layer multipath

The transport-layer solution requires a transport-level protocol that can leverage multiple paths simultaneously. Numerous efforts have been made to design such multipath transport protocols, including R-MTP [42], pTCP [43], mTCP [44], SCTP multihoming [45]. However, the most widely adopted multipath transport protocols are MPTCP [46] and MPQUIC [47].

Recent research in multipath transport has primarily focused on packet scheduling for these protocols, such as OLIA [48], BALIA [49], BLEST [50], DEMS [51], and MPCC [52]. Most of these schedulers focus on bandwidth aggregation, aiming to combine the available bandwidth across all paths under the assumption that each path contributes significant bandwidth. However, in our setting, the low latency channel typically provides only a small fraction of the bandwidth available on the high bandwidth channel (e.g., URLLC bandwidth in 5G is roughly 1% of eMBB bandwidth). As a result, bandwidth aggregation can achieve only very limited performance gains. Aside from

bandwidth aggregation, multipath transport protocols can also be employed to support failover scenarios [53], ensuring resilience when one path becomes unavailable.

2.2.3 Application layer multipath

Applications can participate to further help scheduling traffic across multiple paths. Application layer multipath solutions are often tailored to specific applications, with video delivery being the most common use case. Examples include Xlink [54], Converge [55], Augur [56], and Chorus [57].

There also exist more generic approaches that are not tied to any particular application, and DChannelX falls into this category. However, existing approaches do not explicitly account for or effectively exploit paths with distinct bandwidth–latency trade-offs. Socket Intents [58] and Intentional networking [59] both expose custom APIs to applications and offer OS-level support for managing multiple interfaces. IANS [60], [61] uses socket intents to map a content (i.e., web object in [60] and video chunk in [61]) to specific network path. As we show later, static assignment of application content to paths yields suboptimal performance, since our key insight for effectively utilizing paths with bandwidth–latency trade-offs is to steer traffic at a fine granularity (i.e., at the packet level).

Chapter 3 Accelerating general mobile applications through a network layer steering with DChannel

In this chapter, we explore the use of HBC and LLC for general mobile applications like web browsing. As these apps requires at least a moderate amount of bandwidth, using solely LLC is not feasible. In fact, our experiments emulating web browsing (the most widely used mobile application [62], and far from the most bandwidth-intensive application) over 5G URLLC with 2 Mbps bandwidth show web page load times would be $5.87\times$ worse than with 5G eMBB. These applications will instead be mapped towards eMBB as it provide necessary bandwidth to support variety of general Internet applications.

However, eMBB as an HBC suffers from high and inconsistent latency. This channel will be useful for streaming media but offers little to no improvement for latency-sensitive applications, such as web browsing [25], [63], [64]. Experimentally, web page load time in existing 5G deployments, even in close-to-ideal circumstances (a stationary device and a channel with little utilization), is similar to 4G for pages smaller than 3 MB in size and about 19% faster than 4G for pages larger than 3 MB [25]. This is due to 5G eMBB having 28 ms or larger latency, broadly similar to 4G [25]. Our measurements of 5G mmWave showed similar results, at around 22 ms in ideal conditions.

We then seek to answer this question: Can we simultaneously use LLC and HBC such that latency-sensitive general mobile applications can reap the benefit of the low latency while also enjoy high bandwidth? Mapping an application’s traffic to HBC and LLC is difficult since we have to use LLC’s bandwidth very selectively. Indeed, the main deployed transport-layer mechanism to combine multiple channels, MPTCP [46], assumes two interfaces that are each of significant bandwidth, with the goal of aggregating that bandwidth or supporting failover. LLC’s bandwidth, however, is a rounding error compared to HBC’s. Other works – particularly Socket Intents [58] and TAPS [65] – exploit multi-access connectivity through application-level input and therefore require

application modification. In contrast, our solution in this chapter aims to avoid such inputs to simplify deployment and broaden applicability to future applications.

To solve these problems, we design DChannel, a network layer system that leverages parallel channels to improve the performance of mobile applications. DChannel comprises two modules running at either end of the channels – namely, in the mobile device OS and in a gateway device operated by the service provider. Central to the approach is a packet steering scheme that steer IP packets without requiring any application input. Such fine-grained, per-packet decisions (as opposed to, for example, HTTP object-level steering) are key to making effective use of the limited LLC bandwidth. To decide which packets are worth accelerating, since LLC bandwidth is extremely limited, DChannel treats the channel as an expensive resource and calculates the benefit and cost of utilizing the LLC for each packet. Finally, since the parallel channels could occasionally confuse the transport layer with out-of-order delivery, DChannel employs a reordering buffer in the mobile device and gateway.

To evaluate our design with a concrete scenario, we leverage 5G’s eMBB and URLLC as our HBC and LLC. We evaluate the benefit of DChannel in our experimental testbed (§3.3). Our testbed includes a prototype that can capture and steer application traffic, and a high-fidelity trace-driven network emulator that emulates cellular network latency variability and delay caused by radio resource control (RRC) state transitions [66]. We gather two types of real 5G eMBB traces – mmWave and lowband – in three different scenarios: stationary, low mobility, and high mobility. Our evaluations cover popular web applications such as web browsing and Android mobile applications. Using the testbed, we evaluate our packet steering scheme and compare it with prior approaches such as MPTCP [67] and ASAP [41]. We also evaluate DChannel in live 5G eMBB networks. Our key findings are as follows:

- DChannel, which requires little per-connection state and no application knowledge, yields superior performance compared to the other evaluated schemes—object-level

steering, static packet-size-based steering, as well as prior work, MPTCP and ASAP [41], which used multiple channels in other settings.

- Compared with exclusively utilizing the eMBB, allocating a modest bandwidth of 2 Mbps to URLLC allows DChannel to improve web page load time (PLT). Under conditions that are ideal for eMBB (a stationary client with a line of sight to the base station and full signal strength), DChannel reduces PLT by 20% and 33% in 5G mmWave and low-band settings, respectively. Under more challenging mobile conditions, DChannel improves PLT by 37% and 42% in 5G mmWave and low-band, respectively.
- In addition to web browsing, we evaluated three Android mobile apps in a live environment and find DChannel improves apps responsiveness by 16% on average.
- Somewhat surprisingly, DChannel improves *sustained throughput* in our mobile 5G setting by roughly 10% – a useful side benefit of accelerating the TCP control loop in dynamic environments.

Finally, we discuss deployment strategies, challenges, and future opportunities. We believe our basic techniques can apply to a variety of latency-sensitive applications, and open new opportunities for app developers and cellular providers.

3.1 Background on web browsing traffic

While we evaluate several applications, web browsing is the major focus of DChannel and serves as a running example.

A single web page may contain tens to hundreds of relatively small-sized web objects distributed across multiple servers and domains. Consequently, web browsing traffic is characterized by its often short and bursty flows. A study across Alexa Top 200 pages found that the median number of objects in a page is 30, while the median object size is 17 KB [68]. Fetching these web objects translates to many HTTP request-and-response

interactions across many short flows. The browser fires a page load event when it finishes rendering a page, which is used to determine Page Load Time (PLT), a performance metric for web browsing. Although PLT has some shortcomings, the alternatives are not free from issues, and PLT is most widely used. PLT is typically dominated by DNS lookup, connection establishment, and TCP convergence time—which require little throughput but are highly dependent on RTT. Prior work also showed that increasing TCP throughput beyond ≈ 16 Mbps offers little improvement in PLT [1].

Of course, web page loading is affected by client CPU and server delay, in addition to network delay. Prior work found that 35% of the PLT is spent in client-side computations [69]. But the above characteristics, combined with the fact that mobile CPUs have been getting increasingly powerful [70], still suggest that network latency plays an important part in mobile web performance. Moreover, a significant portion of network latency lies in the “last mile” connection of the cellular network. Many other mobile apps also rely on HTTP-based interaction with cloud services, resulting in similar network performance requirements.

3.2 DChannel Design

3.2.1 High-Level Architecture

To steer application traffic in both uplink and downlink channels, there will be two main components, one in the mobile client device and one in the mobile core network (Figure 2).

On the client, applications interact with the network through a network interface as usual. In our prototype, this is a special virtual TUN interface designated for traffic that should utilize both the HBC and LLC. The client-side agent captures outgoing packets on this interface and implements an algorithm to steer traffic between the two channels. The agent also captures incoming traffic on both channels and merges it into the virtual interface, after buffering it as needed to reorder packets (§3.2.6).

which vary in size and priority. Since LLC is bandwidth constrained, LLC can only deliver small objects faster than HBC.¹ Web pages have complex dependency structures, and certain objects can be on the critical path for web page loading. These critical-path objects need not necessarily be small in size. Small objects might have low priorities such that accelerating them will not improve load time and thus would waste LLC bandwidth. In contrast, high-priority objects can be large such that sending those to LLC will be slower than HBC. Application-level input could help distinguish between these cases.

But object-level splitting has two drawbacks. First, we want to avoid requiring application input, which creates deployment hurdles and extra work for developers. Second, it misses opportunities for latency improvement. A web object that’s not small enough to be sent over LLC will still involve small and latency-sensitive DNS lookups, TCP connection establishment, TLS handshaking, and ACKs. Accelerating this traffic could significantly reduce object delivery time. We later demonstrate (§3.4.3) that object-level splitting is less effective than finer-grained packet-level steering.

Steering packets at the network layer (e.g., IP datagrams) comes with its own challenges, however. First, we do not have any application-level insight into the flow: we do not necessarily know how packet-level acceleration affects application-level acceleration, so we will need a careful steering heuristic. Second, even if we identify the packets to accelerate, sending packets within a flow across two different channels might result in the packets arriving out-of-order, confusing TCP. To address this issue, we will introduce a small *reordering buffer (ROB)* at the endpoints. The following subsections discuss these components of the design.

3.2.3 Packet Steering Intuition

Define a “message” as a sequence of one or more packets such that the receiving endpoint can take some useful action after receiving the full message. For example, an individual

¹If URLLC is assigned a capacity of 2 Mbps (≈ 250 bytes per ms) and its RTT is ≈ 15 ms less than that of eMBB, any object of size larger than 3.75 KB are likely to be delivered faster on eMBB.

SYN or ACK is a message (because the transport layer can act on it), and an HTTP request or a full response spread across multiple packets is a message (because the application may be able to process the request, display an object to the user, etc.). In contrast, an individual data packet belonging to a large HTTP request/response is not a message on its own and would not be worth accelerating individually since we need to accelerate the whole sequence of packets to finish the message.

Ideally, we would like to accelerate the delivery of messages, especially those that are most valuable to accelerate, within the bandwidth constraints of the LLC. This suggests a *cost-rewards calculation* weighing the benefit of accelerating a message against the cost of utilizing the meager bandwidth of the LLC which might be better spent on other messages.

A direct, exact cost-rewards calculation is infeasible since DChannel running at the network layer lacks full knowledge of message boundaries (in the application’s data stream), as well as the relative value of messages to the receiver’s transport layer or application. This leads us to begin with a permissive assumption: any packet *might* be a message boundary and we will optimistically consider accelerating it. Nevertheless, even operating transparently at the network layer, DChannel does have certain information about rewards and costs that will help it distinguish among packets.

First, the benefit of steering a packet to the LLC depends on how much its arrival time would improve, if at all, compared to using the HBC. This depends on packet size, current output queue lengths for both channels (which are locally observable), and latency of both channels (which can be estimated). In addition, the vast majority of applications utilize TCP or other transport that delivers messages in order.² This means that for a message inside packet P_i , *delivery of the message to the application* (as opposed to the delivery of P_i to the receiving host) will depend not only on the arrival time of P_i , but also on the arrival time of packets $P_0, \dots, P_{i-1}$ (which can also be estimated). For

²Some don’t, of course, but our goal in this work is to develop generic packet steering, leaving application-specialized schemes for the future.

example, suppose P_{i-1} was sent over the HBC, and P_i is ready to send immediately after. If P_i is also sent over HBC, the pair will arrive at about the same time. If P_i is sent over LLC, it will very likely arrive much sooner, but will end up waiting for P_{i-1} before it can be delivered to the application, meaning sending over the LLC is likely not useful in this case.

Second, the cost of utilizing LLC resources will depend on the packet length and how much the LLC will be in demand for other messages in the near future. The latter is not perfectly known, but current or recent outgoing LLC queue depths provide some signal.

The net effect of the above considerations is that packets should tend to get steered to the LLC when they are smaller, and when they are more isolated in time as individual packets or members of short packet sequences. This corresponds well with the intuition of prioritizing acceleration of control messages or small application-level messages. We now proceed to describe how we realize this cost-rewards approach.

3.2.4 Rewards and Cost

Problem statement. The packet steering algorithm is presented with a sequence of packets and needs to decide if each packet P_n should be sent via LLC or HBC. We let $P_1, \dots, P_n$ denote the sequence of packets in a single end-to-end flow (by which we mean a unidirectional transport layer connection, which may contain multiple messages).

Rewards. At the packet level, the objective is to minimize the *packet completion time* C_n , defined as the time by which all packets $P_0, \dots, P_n$ would arrive at the receiver. This captures the intuition (§3.2.3) that any P_n might be a useful message to accelerate on its own, but it wouldn't be delivered to the application until prior packets are also delivered. The benefit of sending a packet P_n via LLC is thus the reduction of C_n if P_n is sent via LLC (denoted $C_{n,LLC}$), compared to when it is sent via HBC (denoted $C_{n,HBC}$). Thus, we calculate the rewards for sending P_n via LLC as:

$$R(P_n) = C_{n,HBC} - C_{n,LLC} \quad (1)$$

To calculate the above, we first need to estimate the delivery time D for a packet that depends on the channel/link³ propagation delay $Dprop_{link}$ and bandwidth B_{link} , packet size, and the link's queue size Q_{link} at time t_n . The Q_{link} counts the number of bytes that have been enqueued for transmission through a *link* but have not yet been transmitted out the interface. Delivery time for P_n on a certain *link* is thus:

$$D_{link}(P_n) = Dprop_{link} + (size(P_n) + herl)/B_{link} \quad (2)$$

The packet completion time for P_n (C_n) should also account for completion times of P_0 through P_{n-1} (i.e., C_{n-1}) since P_n may arrive at the receiver before P_{n-1} , especially if P_n is sent over LLC and P_{n-1} was sent over HBC. Thus, we can calculate ($C_{n,link}$) as:

$$C_{n,link} = max(C_{n-1}, (t_n + D_{link}(P_n))) \quad (3)$$

Note that $Dprop_{link}$ are nondeterministic, comprising dynamic channel delay and any congestion along the channel's path, and will thus have to be estimated. We return to this later.

Cost. The cost of sending a packet to the LLC comes from the increased utilization of LLC. Intuitively, the cost should increase with the added queueing delay that a packet arriving very soon after P_n would experience, i.e., $size(P_n)/B_{llc}$. The cost should also be higher if the LLC is currently more highly utilized so that its limited capacity is reserved for higher-reward packets. We use a heuristic that captures this by adding these two

³We use these terms interchangeably for convenience. Note, however, the LLC channel may involve acceleration in the WAN in addition to the RAN, so it actually may span multiple physical links.

effects; specifically, we compute the cost (or fare F) of putting P_n on LLC as:

$$F(P_n) = (size(P_n) + Q_{llc}(t_n))/B_{llc} \quad (4)$$

Note that to be more precise, we should compute the difference in costs of putting the packet on LLC vs. HBC. But as the HBC bandwidth is dramatically higher, its cost is negligible and we omit it for simplicity.

Comparing rewards and cost. At a high level, we want to steer packets to LLC when the rewards outweigh the costs, but comparing them involves a tradeoff: the benefit is immediate to packet P_n , whereas the cost affects possible subsequent packets which may not appear. We introduce a parameter α to capture this, so that we will send a packet to LLC when:

$$R(P_n) > \alpha F(P_n) \quad (5)$$

Calibrating α . If we set α too low, a flow may aggressively send packets to LLC so that it will deny resources to another flow in a multi-flow application. If we set it too high, we can be too conservative in utilizing the fast LLC. To find a good α and determine how sensitive performance is to its value, we conduct experiments with web browsing across different alpha values. We load 40 web pages from our corpus over different α values and pick α with the best Page Load Time (PLT) result on average. We use our testbed (§3.4.1) and apply the packet steering over HBC and LLC. For LLC, we use 5G NR URLLC as a reference where the RTT and bandwidth is 5 ms and 2 Mbps. For HBC, we vary its RTT while fixing bandwidth at 200 Mbps.

The detailed results are shown in Table 1. In summary, the results confirm that setting α too low or high has suboptimal performance. The best value for HBC RTT of 20 ms to 60 ms is 0.75. This RTT range covers most cases of 5G eMBB. As the RTT increases to 80 ms and higher, $\alpha = 1$ is slightly better. The difference, however, is less

Table 1: Percentage of improvement or ‘speedup’ in PLT (%ps.) along with the percentage of bytes (%sz.) sent via URLLC for various values of the eMBB channel RTT. The table also shows how the different values of α affect the performance benefits.

RTT (ms)	$\alpha = 0$		$\alpha = 0.25$		$\alpha = 0.5$		$\alpha = 0.75$		$\alpha = 1$		$\alpha = 2$		$\alpha = 3$	
	%ps.	%sz.	%ps.	%sz.	%ps.	%sz.	%ps.	%sz.	%ps.	%sz.	%ps.	%sz.	%ps.	%sz.
20	16.7	13.3	16.6	10.7	18.4	5.9	18.9	5.7	18.5	5.5	15.6	4.7	14.2	4.1
40	31.1	18.8	33.2	16.2	34.0	13.9	34.7	11.9	34.0	11.1	32.8	8.5	31.8	5.8
60	37.5	22.6	40.9	19.5	41.2	17.4	42.1	14.9	40.8	14.2	40.6	11.1	37.4	9.8
80	43.2	26.3	46.3	22.9	46.3	19.9	46.3	17.7	47.1	16.5	45.6	13.3	45.1	11.6
100	47.1	29.4	48.6	26.1	49.7	22.5	50.2	19.9	50.7	18.3	49.8	14.9	48.5	13.1

than 1%. We use $\alpha = 0.75$ for all subsequent experiments.

Note on design. The steering approach described here is not an optimal choice derived from a model – it is a heuristic, particularly the calculation of cost and calibration of α , in part since some of the relevant information (like the application-level importance of a particular packet) is unavailable. However: (1) we find the heuristic does perform well in realistic environments, (2) even if poor decisions do occur, they lead only to suboptimal performance, rather than a correctness problem, and (3) performance is not very sensitive to the exact value of α . In particular, even with $\alpha = 0$ – which corresponds to the greedy strategy, where each packet uses LLC whenever it expects a reward for itself – there is still a very good PLT improvement, within 5% or less of the best α . That said, this problem could be interesting to formalize in the future, perhaps as an online algorithm that could provide worst-case guarantees, or using queueing-theoretic tools.

3.2.5 The Packet Steering Algorithm

Putting together the above pieces, the complete steering algorithm is shown in Algorithm 1. To make a decision, the algorithm requires (1) packet size, (2) current LLC queue size, (3) LLC bandwidth, and (4) latency of both LLC and HBC. The LLC bandwidth is controlled (assigned by the operator) so it is known, and (1) is directly observable.

LLC queue size (2) may directly be observable at the client, assuming its NIC is

Algorithm 1: DChannel steering algorithm

Result: Send a packet to either HBC or LLC

$c_llc = t_now + llc_prop_delay + (size(packet) + queue_llc) / bandwidth_llc;$

$c_hbc = t_now + hbc_prop_delay;$

$rewards = c_hbc - \max(prev_c, c_llc);$

$cost = (size(packet) + queue_llc) / bandwidth_llc;$

if $rewards > \alpha * cost$ **then**

 send(pkt, LLC);

$prev_c = \max(prev_c, c_llc);$

else

 send(pkt, HBC);

$prev_c = \max(prev_c, c_hbc);$

end

limited to the LLC bandwidth. But the proxy may have a higher local NIC rate. The proxy, therefore, tracks outgoing traffic per user and computes what the queue depth would be if the NIC had been limited, by dividing the outgoing traffic with the dequeue rate of the path (i.e., estimated bandwidth of the path). Depending on the service provider’s admission control policy, the rate could alternately be explicitly limited at the proxy. Client can also apply similar approach if (2) is not directly observable. In Algorithm 1, we omitted the HBC queueing delay caused by $(size(P_n) + Q_{hbc}(t_n))/B_{hbc}$, since HBC bandwidth (B_{hbc}) is typically large and relatively hard to measure.

Latency (4) has to be estimated. To do this, we perform periodic handshakes (e.g., in every 500 ms in our use case). The handshakes consist of four steps, all with UDP packets: (1) the client agent sends a special packet we call a “D-SYN” to the proxy agent using both HBC and LLC. (2) The proxy agent upon seeing a D-SYN responds with “D-SYN/ACK” packets sent across both HBC and LLC. (3) The client agent receives the D-SYN/ACK packets, updates the base RTT value for both channels based on the difference between D-SYN/ACK receive time and D-SYN release time, and replies with “D-ACK” packets sent across both channels. (4) The proxy agent receives the D-ACK packets and updates the base RTT value for both channels. We use the minimum RTT value for the measurement. As we will see in the evaluation (§3.4), very rough latency

estimates are sufficient.

The algorithm requires maintaining per-flow state, specifically to store C_{n-1} , the estimated completion time of the most recent previous packet. The proxy also stores per-user state for its queue depth calculation.

3.2.6 Reordering buffers at the endpoints

Splitting packets across asymmetric paths (particularly with a latency differential, as there is for LLC vs HBC) can cause out-of-order packet delivery, which can be harmful to application performance. In particular, TCP uses out-of-order packets as a signal of congestion, potentially causing retransmissions and a drop in sending rate. To solve this problem, we adopt a reordering buffer (ROB) in the receiving direction of each of our agents, to buffer packets arriving only from LLC. Note that we only buffer packets arrived from LLC as we only want to handle packet reordering caused by sending packets through the faster LLC and not to solve reordering caused by external factors such as wireless losses.

To avoid unbounded buffering delay if the previous packet was lost, the ROB also releases packets after a timeout. Ideally, the timeout should equal the latency of HBC, but because the latency of HBC can be variable and hard to track, we use a conservative 100 ms timeout. We evaluate the effectiveness of this timeout value under random packet loss in §3.4.

3.3 Prototype and Experimental Setup

Our experiments involve a client representing a mobile end-user application (e.g., a web browser) fetching content from a web or content server. Both the client and server endpoints have access to two interfaces, one representing the high-bandwidth channel (HBC) and the other the low-latency channel (LLC). In the case of 5G, HBC and LLC map to eMBB and URLLC, respectively. Depending on the experiment conditions, the

interfaces may be real or emulated. We masked the two interfaces at the endpoints, however, using a smart DChannel virtual interface implemented on top of a TUN device; the client and server use only this virtual interface to send and receive data. Our DChannel prototype then performs endpoint-transparent (and application-agnostic) steering of traffic.

We developed a DChannel prototype and packaged it as a UNIX shell, similar to the shells in Mahimahi [72]. The shell captures all outgoing traffic from any *unmodified* application running within it and tunnels them to our DChannel implementation; it processes incoming traffic in a similar application-transparent manner, so both the steering and buffering modules of DChannel are used. Our DChannel prototype attaches additional metadata (sequence number and flow ID) prior to transmission to assist the receiver in reordering packets and strips this before delivering to the application. We used our own metadata header as a convenience, but in a real implementation, this could be avoided by looking inside the layer 4 header.

We evaluated the performance of DChannel using this prototype under two settings. The first is a *live* setting where we used the actual 5G NR eMBB channel as HBC. The second setting, in contrast, is one where we emulated the eMBB channel based on traces that we gathered from an actual 5G eMBB channel. In both settings, since URLLC is not yet commercially available, we emulated its “expected” behavior (based on the 5G specification [35]) using a low-latency, bandwidth-limited wired Ethernet connection.

3.3.1 Live-eMBB Setting

In this setting, DChannel steers traffic over two real interfaces (Fig. 3): One interface is tethered with a 5G phone for providing access to a *live* eMBB channel, while another is connected to a low-latency bandwidth-limited Ethernet connection for emulating the URLLC channel. Packets transmitted over the 5G eMBB channel traverse the core network of the mobile provider before exiting via the packet gateway (i.e., *mobile path*)

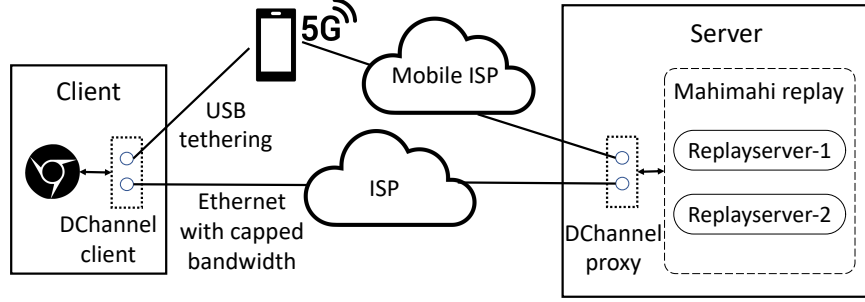


Figure 3: In our live 5G eMBB testbed, the client has two paths to the server: One path over a tethered connection to a 5G phone for utilizing the eMBB channel, and the other through a bandwidth-capped connection over Ethernet, for emulating the URLLC channel.

and then one or more ASes in the public Internet (i.e., *Internet path*) to reach our server. Data sent over the Ethernet interface, in contrast, traverse a traditional ISP and then one or more ASes to reach the server. On the server side, DChannel receives all the packets from both the interfaces, reorders them (if required), and then delivers them to the server-side application via the TUN device.

We used Ethernet and not WiFi for emulating URLLC, since the channel is expected to provide high reliability (≥ 0.9999) [12]. We capped the bandwidth of this link using `netem` to emulate the low bandwidth of URLLC. Since the client must remain physically plugged in to a wired network for emulating URLLC, this setting allows us to study performance only in stationary conditions.

3.3.2 Emulated-eMBB Setting

To evaluate DChannel under a wide variety of scenarios, specifically those including client mobility, we used *trace-driven* emulations. Below, we describe how we captured the network (latency and bandwidth) traces of the 5G eMBB channel under stationary and low-to-moderate mobility scenarios and used them in our emulations.

Collecting network traces. To capture the temporal variability of mobile networks, we measured both the latency and throughput of the eMBB channel over time.

Table 2: Characteristics of network traces gathered from actual 5G deployments at different locations and under different conditions. ‘p50’ and ‘p98’ refer to the 50th and 98th percentiles, and ‘CV’ refers to the coefficient of variation.

Trace name	Span (mins.)	RTT (ms)					Mean bw. u/d (Mbps)	Description
		min.	p50	p98	mean	CV		
<i>mmWave-Stationary (MM-S)</i>	60	18	22	106	29.88	0.77	60/140	UE was in a building in the downtown Chicago, placed near a window with a base station in line of sight.
<i>mmWave-Walking (MM-W)</i>	56	16	22	120	30.32	0.98	60/110	UE was held by user walking in downtown area of Chicago.
<i>mmWave-Driving (MM-D)</i>	18	18	40	236	56.15	0.96	60/100	Phone was with a user driving through the downtown area of Chicago at low to moderate driving speeds.
<i>LowBand-Stationary (LB-S)</i>	60	34	40	132	45.20	0.50	26/93	Phone was located in a building in a university campus. It was placed near a window with full signal strength.
<i>LowBand-Walking (LB-W)</i>	53	32	52	156	58.94	0.50	21/63	Phone held by user walking in a university campus.
<i>LowBand-Driving (LB-D)</i>	23	34	54	202	68.84	0.62	15/57	Phone was with a user driving near a university campus.

Latency traces. We measured the latency of the eMBB channel by periodically sending probes (UDP packets) from the client to the server. We set the probing period to 15 ms to force the UE radio to remain always in “active” mode and generate only a small amount of probe traffic to avoid queuing. Our measurements capture the latency imposed by the base station and core network, since our server was always in close proximity to the client (i.e., less than 150 miles), minimizing the Internet-path latency. Our `tracert` from the client to the server, although not shown in the paper, also confirmed that the latency between the client and the server was very close to the latency between the client and the packet gateway.

Bandwidth traces. We measured the throughput across time of both uplink and downlink channels by saturating them with MTU-sized UDP packets. Since TCP cannot

reliably saturate the highly variable cellular uplink and downlink concurrently, we used an overestimated fixed sending rate to always fill the queue. First, we measured the maximum supported upload and download UDP throughput using existing tools such as `iperf`. Then, we sent traffic at this maximum rate from both endpoints. Finally, we used the actual packets received over time by the endpoints to estimate the uplink and downlink capacities.

Measuring both latency and bandwidth. A key challenge in measuring both latency and bandwidth simultaneously is avoiding interference: bandwidth-intensive operations can saturate the link and fill the queue, thereby inflating the latency. Since cellular networks use per-user queues, we addressed this challenge by measuring latency and bandwidth from separate devices. When using two separate devices, we did not see any perceivable interference for measurements on 5G low-band, although we observed them on 5G mmWave. Specifically, we observed inflation in latency if a nearby device was uploading data at more than 5 Mbps using mmWave.⁴ For 5G mmWave, we measured, hence, only the downlink throughput over time; we set the uplink bandwidth to a single, fixed rate of 60 Mbps.

The accuracy of temporal variations in latency matters *most* for our trace-driven emulations, since the main applications that we use in our evaluations, web browsing and mobile apps, are latency-sensitive. The performance of such applications crucially depends on TCP-related configurations (e.g., initial congestion window) and network latency (or RTT) rather than on available bandwidth, particularly when the bandwidth is more than 16 Mbps [1]. Our approach to estimating bandwidths, therefore, is adequate for our evaluations.

⁴Low-band uses OFDMA so multiple devices can communicate at the same time and the latency is not inflated, while mmWave uses single carrier modulation, where multiple devices must take turns transmitting and the antenna must switch its beam pattern.

Emulating the traces. In the emulated-eMBB setting, we run both the client and the server on the same machine. DChannel then steers traffic between them over two virtual interfaces, emulated using an extended version of Mahimahi [72]. Specifically, we extended Mahimahi’s delay shell to vary the eMBB channel latency over time, based on a trace generated from a real 5G deployment. The modified delay shell accepts a trace comprising a “timeline” of RTT values and halves each value to derive the individual uplink and downlink latency timelines. The shell then assigns per-packet latency by choosing an uplink or downlink latency by matching the time a packet arrives at the interface against the timelines. Since the trace-file granularity is one RTT sample per 15 ms, we use linear interpolation for assigning RTTs arriving between two samples. Similarly, we emulated URLLC with a propagation delay of 5 ms and bandwidth of 2 Mbps, unless noted otherwise.

Mobile applications’ traffic (especially web browsing) is typically bursty in nature and contains periods of inactivity. To preserve energy during *idle* periods, UEs switch to a low-power (or “sleep”) state, which supports discontinuous reception (DRX). The transition to the low-power state depends on an *inactivity timer* that we observed (through probing [64]) to be around 30 ms for 5G mmWave; once the device enters this state, it will “wake up” periodically (every 40 ms). When emulating the latency traces, we therefore also estimate the radio power states of the device (based on its activity) and take into account any additional latency the state transitions may impose. A packet that arrives 20 ms after the UE enters the sleep state, for instance, will experience an additional 20 ms delay before it is processed. This delay, however, is not incurred on the uplink. For 5G low-band, we set the inactivity timer to 100 ms and wake-up interval to 20 ms.

For the bandwidth emulation, we extended Mahimahi’s link shell to emulate a time-varying bandwidth that changes every second. To emulate a link of capacity 60 Mbps at time n seconds, for instance, this extended link shell will release 7.5 KB per

millisecond. In our emulation tests, we also used a FIFO (drop-tail) queue, and we set the buffer to 800 MTU-sized packets.

3.4 Evaluation

We evaluated DChannel using 5G eMBB and URLLC as HBC and LLC, respectively. We ran the client (e.g., a web browser) on a laptop, unless otherwise mentioned. The laptop had 16 GB RAM, 512 GB SSD, and an Intel Core i7 processor running Ubuntu 20.04 (Focal Fossa).

3.4.1 Testbed Configuration

In the live-eMBB setting, we tethered the laptop with a Google Pixel 5 phone using USB (refer Fig. 3). We ran the live experiments from two locations: UIUC campus with access to 5G low-band and the Chicago downtown area for 5G mmWave access. We emulated the URLLC link between the client and server using a wired (Ethernet) link and configured it based on URLLC end-to-end specification and use-cases [35]. The emulated link provides 5 ms RTT between the client and the network gateway and has 2 Mbps capacity. At the 5G mmWave test site, however, the wired link only provided a minimum latency of 8 ms for the URLLC emulation.

We also collected latency and bandwidth traces (summarized in Tab. 2) at the two test locations under three mobility conditions: stationary, walking, and driving. All traces were collected using Google Pixel 5 phones with Verizon 5G. Though mmWave offers lower latency (for eMBB) than low-band, it also experiences higher variance than low-band, even when the UE was stationary. This inconsistency in performance becomes even worse under mobility. Low-band offers a stable, albeit relatively higher, RTT than mmWave. We ran all the components, i.e., the client, DChannel’s modules, and the server, on the laptop in the emulated-eMBB setting, and used the traces (in Table 2) for emulating the eMBB channel.

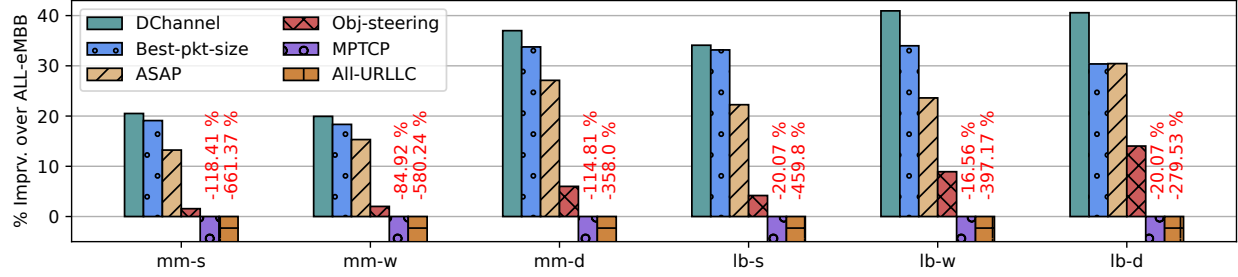


Figure 4: DChannel offers at least 20% lower PLT compared to that of All-eMBB, and it performs better than all other schemes. MPTCP’s PLTs are 17% to 118% worse than when using a single eMBB channel across all traces.

3.4.2 Application use cases

We evaluated DChannel on 5G under a wide variety of network conditions to highlight its benefits for web browsing and web-based mobile (Android) applications. We supplemented these experiments with a bulk-download application for demonstrating DChannel’s merits for long (i.e., bandwidth-intensive) flows.

Web browsing. To measure the improvements brought about by DChannel for web browsing, we first fetched a set of 200 web pages of “popular” websites, selected uniformly at random from the Hispar corpus [73]. The sample comprised 40% of landing and 60% of internal pages from 200 websites. The median web page size and the number of objects are 3.7MB and 60, while the 95th percentile are 11.8MB and 168. When fetching these pages, we recorded all the HTTP requests and responses using mitmproxy [74]. Then we used a version of Mahimahi with HTTP/2 support [75] to serve the responses from our server. While recording the pages, we also estimated the server response time for each request by subtracting the *time-to-first-byte (TTFB)* from the client-to-server RTT. We used the server-response times to emulate server-side processing delays during the replays.

We used an unmodified Chromium browser spawned within a DChannel shell to fetch the pages from our server. We cleared the browser and DNS caches prior to each fetch

and used the default Linux congestion control, TCP CUBIC, unless noted otherwise, for all web-browsing experiments. We measured the *page-load time (PLT)*, based on the `onLoad` event [76] in each experiment, on each fetch. In the live-eMBB setting, we first used the DChannel scheme to fetch a page and repeated that page fetch in quick succession using a different scheme. We calculated the difference in PLT between the different schemes and repeat the fetch five times to compute the mean difference in PLTs. In the emulated-eMBB setting, we picked a random sub-sequence from a trace for each page fetch. Given a page, we used the same sub-sequence for measuring the PLT across different schemes and computed the difference in PLTs. We used the mean of the PLT differences across five trials, with each using a random sub-sequence of the trace, for comparing and contrasting the different schemes.

Mobile application. We downloaded three popular Android applications from the Google play store, one each from three categories: social (Reddit), shopping (eBay), and news (CNN). We ran the applications on the phone (Google Pixel 5), but tunneled all network traffic to a DChannel shell running on our laptop. We then used the DChannel to steer the traffic over the appropriate channels. We evaluated the mobile applications only under the emulated-eMBB setting. Note that this setup may underestimate the performance improvements because of the additional overheads in tunneling the traffic from the phone to the laptop. We measured the RTT from our laptop to the application server to be 12 to 30 ms, which is significant given that our emulated URLLC and eMBB RTT can go as low as 5 ms and 16 ms.

To evaluate application performance, we calculated the *interaction response time (IRT)* [77]. IRT measures the time elapsed between when a user performs an interaction to when the end screen for that interaction is completely rendered. We automated user interactions with `AndroidViewClient` [78] and cleared application caches within each trial. We recorded the phone screen using `FFmpeg` [79] during each interaction and identified

when the screen stopped changing visually using `scenecut-extractor` [80]. Since we do not control the servers used by the application, we repeated the interaction experiments with DChannel and other schemes, one after the other, in quick succession. We used the median IRT across ten trials to compare the performance of different schemes.

Bulk download. The setup for this use case is similar to that for web browsing. We simply used `curl` as the client to download a file hosted in our (Mahimahi) server and repeated the download five times.

3.4.3 Comparing steering schemes

We compared the PLT of DChannel scheme with that of five other schemes across multiple scenarios (using the traces in Tab. 2) in the emulated-eMBB setting.

The other schemes are as follows. **All-URLLC** steers all traffic over URLLC. **Obj-steering** requests web objects (requests and responses) on URLLC whenever its fetching time is smaller than eMBB. **Best-pkt-size** steers packets whose size is lower than the *best* predefined threshold to URLLC. **MPTCP** uses the two channels concurrently, by running the Linux kernel implementation of MPTCP [67] with the default configuration. **ASAP** [41] was designed for satellite networks. It diverts traffic from satellite links (or eMBB in our case) to 3G or 4G (or URLLC) since the latter has lower latency with a higher cost per byte than the former. ASAP code is not publicly available, so we used our in-house version.

Fig. 4 compares the relative difference in PLT of each scheme compared to that obtained when only using eMBB (i.e., All-eMBB). DChannel offers the best performance under all network conditions in our emulations. In stationary scenarios (MM-S and LB-S), it improves PLT by about 20% when using 5G mmWave and 35% with 5G low-band. These improvements in 5G mmWave and low-band correspond to absolute reductions in PLTs of about 290 ms and 642 ms, respectively, compared to the All-eMBB

Table 3: Comparing the performance of DChannel with All-eMBB and All-URLLC when fetching the (182 KB) landing page of `amazon.com`.

Perf. metric	All-eMBB	All-URLLC	DChannel
<i>DNS lookup (ms)</i>	44	8	8
<i>TCP connect (ms)</i>	42	6	6
<i>TLS connect (ms)</i>	53	30	30
<i>Object transfer (ms)</i>	209	809	144
<i>Total load time (ms)</i>	349	853	189

scheme. Per Fig. 4, DChannel’s performance increases in scenarios that involve UE mobility (MM-W, MM-D, LB-W, and LB-D). The PLT improvements are higher for low-band than mmWave since the former exhibits higher latency than mmWave (refer Tab. 2). We examined the relation between DChannel’s performance and eMBB latency in detail in §3.4.4.

Where do DChannel’s gains come from? To illustrate the answer to this question, we used `curl` to fetch the landing page (i.e., the root document, “/”) of `amazon.com` in the emulated-eMBB setting (40 ms RTT and 200 Mbps) without any server-side response delay. DChannel performs better than not only all-eMBB, but also all-URLLC (Tab. 3). While URLLC has lower latency than eMBB, it also has a significantly lower bandwidth than eMBB. We identified three key sources of performance gains by analyzing DChannel’s per-packet decisions. DChannel steers three types of packets over URLLC: (1) DNS packets; (2) *control* packets (e.g., SYN and client-to-server ACK packets); and (3) small *data* packets. Sending DNS and SYN packets over URLLC reduces DNS lookup and TCP connection-setup times, and accelerating ACK packets reduces the object transfer time. The last category includes small data transfers such as the TLS client-key exchange and HTTP requests.

Packet vs. web objects steering. Obj-steering performs HTTP request and response on URLLC when the web object size is small such that it will finish faster than eMBB.

The scheme only offers slight improvement to PLT (2-14%). This is because not all small objects are critical. In fact, we found out that only 14% of small web objects have VeryHigh priority [81].

DChannel vs. static packet-size-based steering. Best-pkt-size steers individual IP packets whose size is smaller than the *best* static threshold to URLLC, and the reordering buffer will reorder any out-of-order packets. To find the best threshold, we performed web page load experiments for each trace with five different (size) thresholds: 250, 500, 750, 1000, 1250, and 1400 bytes. We found 750 bytes and 1000 bytes give the best-averaged result (across the stationary, walking, and driving scenarios) for mmWave and Low-Band traces, respectively.

DChannel shows an overall better improvement than best-pkt-size across different network conditions, albeit best-pkt-size offers similar improvements in stationary traces. In stationary traces, network latency is more predictable, and static decisions might suffice. When the network conditions are more variable, such as in the driving scenario, however, the static decisions do not suffice. DChannel observes network conditions, as they evolve, and estimates URLLC channel usage to make steering decisions dynamically. DChannel will not steer small packets to URLLC, for instance, if the channel is already congested. Note that these results are overly generous to best-pkt-size, for comparison purposes: the best size is selected in retrospect after running on the test scenarios. In reality, determining a single packet size threshold would be complicated: it depends on application traffic patterns as well as network conditions.

Is MPTCP not designed to exploit multiple channels? MPTCP works at the transport layer, and in general, it load-balances application traffic among the available paths and aggregates their throughput. In our evaluations, MPTCP performs worse (by inflating PLTs between 16% and 66% across different conditions⁵) than simply using only

⁵We clipped the bottom of the Y-axis in Fig. 4 to focus on performance gains.

the eMBB path. This poor performance stems from MPTCP’s default scheduler (minRTT), which prefers the path with the smallest estimated RTT. This scheduler thus infers that URLLC is better than eMBB and diverts traffic to URLLC until experiencing congestion. DChannel, unlike MPTCP, works at the network layer such that it allows steering data packets on eMBB and ACK packets on URLLC. MPTCP cannot perform such packet-level steering, since it results in each path having a separate data-ACK loop, which MPTCP cannot support.

DChannel vs ASAP. ASAP identifies the different phases of a web transaction (e.g., TLS handshake and HTTP request) and accelerates packets of latency-sensitive phases. It accelerates, for instance, TLS/SSL handshake as well as HTTP request traffic, but leaves HTTP responses to eMBB. ASAP performs better than all other schemes except DChannel. It falls behind DChannel, however, because of its static heuristics (e.g., accelerate all HTTP requests). HTTP requests are typically, but not always, small. A user uploading a photo, for instance, is one example where the assumption fails to hold. ASAP also encounters problems when the user browses complex internal pages that push some data to the server.

3.4.4 Understanding DChannel performance

Below, we investigated how DChannel performs under tightly controlled network variables. We used a fixed network latency and bandwidth for the experiments below.

Performance under high eMBB RTT. We evaluated DChannel under eMBB RTT inflation and found it to be resilient towards the RTT increase (Fig. 5c). Specifically, DChannel is $2\times$ faster than the baseline when eMBB RTT is held at 100 ms, which is possible in device mobility as Tab. 2 shows. As eMBB RTT increases, DChannel web PLT degrades at a slower rate compared to All-eMBB because it uses eMBB primarily for bandwidth-sensitive (low rewards) traffic that is not affected as severely by the

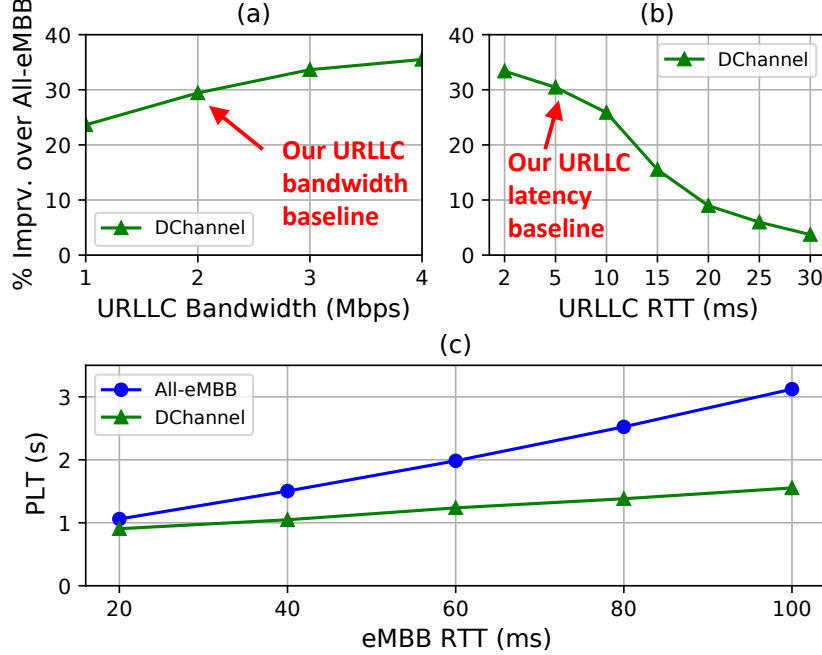


Figure 5: DChannel under varying eMBB and URLLC network conditions, conducted using the baseline (RTT (ms) / bandwidth (Mbps) for eMBB and URLLC are 40 / 200 and 5 / 2) with a single varying parameter.

increased latency. Inline with our trace-based evaluation under mobility, the parallel channel setup can be extremely effective when the eMBB channel quality degrades.

Varying URLLC latency. Although URLLC is expected to deliver consistent low latency, we evaluated latency inflation on URLLC to reflect scenarios where web traffic is de-prioritized in favor of critical traffic. DChannel is still superior even when URLLC latency increases up to 30 ms and eMBB held at 40 ms (Fig. 5b). As expected, as URLLC latency increases and becomes closer to eMBB, the PLT improvement rapidly diminished because LLC no longer offers a competitive resource despite its higher cost.

To evaluate DChannel sensitivity to URLLC latency instability, we changed URLLC propagation latency over time to random values that follow Gaussian distribution. We kept the URLLC mean to 10 ms and ran experiments with an increasing amount of variation, controlled with the Gaussian distribution's coefficient of variation (CoV). We

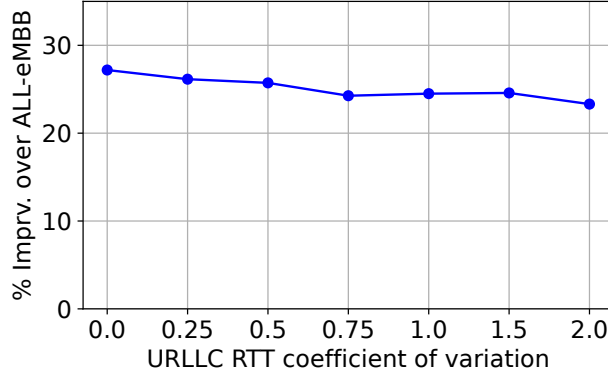


Figure 6: DChannel PLT improvement under time-varying URLLC RTT randomly generated according to a Gaussian distribution.

set the URLLC bandwidth to 2 Mbps while the eMBB RTT and bandwidth are 40 ms and 200 Mbps. We found that DChannel performance is relatively robust to the URLLC latency change (Figure 6); the PLT improvement only drops from 27.18% to 23.31% when the URLLC latency CoV changes from 0 to 2.

Varying URLLC bandwidth. The URLLC bandwidth is generally limited to ensure its reliable and low latency service. We tested DChannel under varying URLLC and summarize that its improvement flattens as URLLC bandwidth increases past 2 Mbps (Fig. 5a). As URLLC bandwidth increases, DChannel aggressively offloads packets belonging to a larger transfer (e.g., HTTP response) to URLLC. It may not affect the completion time, however, as it still needs to wait for the remaining part to be transferred over the eMBB.

Working with Incorrect Latency Estimates. DChannel requires estimates of eMBB and URLLC latency to calculate rewards and cost (§3.2.4). While URLLC latency is predictable, it can be hard to get an accurate measurement of eMBB latency, especially under mobility. To better understand the sensitivity of DChannel to the latency estimates, we evaluated DChannel under underestimated and overestimated latency. Fig. 7 shows that underestimating eMBB latency is safer: When DChannel

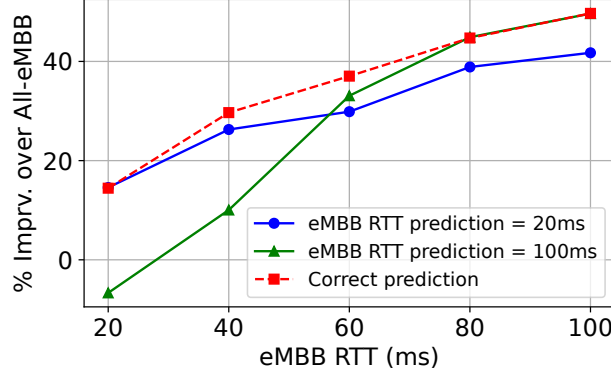


Figure 7: Effect of underestimating and overestimating eMBB latency on the mean PLT improvement.

Table 4: Mean PLT with TCP BBR under different eMBB RTTs. The URLLC RTT is set to 5 ms

	RTT=20ms	RTT=100ms
All-eMBB	915 ms	2661 ms
DChannel	716 ms (21%)	2713 ms (-2%)
pkt-uplink	860 ms (6%)	1628 ms (39%)

underestimates eMBB latency as 20 ms (from 100 ms, which is $5\times$ higher), the PLT improvement only decreases by 8%. Underestimating eMBB latency will underestimate the rewards, causing a more conservative use of URLLC and ensuring that the offloaded packets are high rewards packets. Overestimating the latency will, in contrast, overestimate the rewards, resulting in unnecessary packets being offloaded to URLLC, and slowing down the channel. Per Fig. 7, overestimating eMBB latency by $5\times$, DChannel gets worse performance than the baseline.

DChannel under TCP BBR. Since DChannel steers packets over two channels, the sender may notice an abrupt change in the flow RTT. We evaluated DChannel in TCP BBR, which uses RTT measurement to determine whether a path is congested. Tab. 4 shows the result when low (RTT=20 ms) and high (RTT=100 ms) eMBB latency are used. When the eMBB RTT is 20 ms, BBR works perfectly with DChannel because eMBB and URLLC latencies are not that different. As the latency gap widens, however,

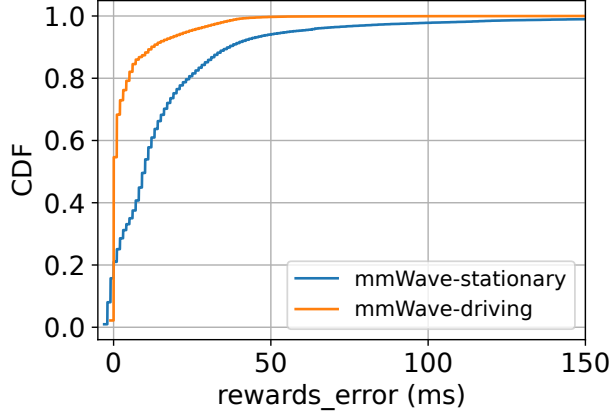


Figure 8: DChannel calculates rewards error distribution ($R_{real} - R_{est}$) across individual packets. A positive value denotes R underestimation. We limited the x-axis due to the high rewards error in the tail (155 ms and 604 ms at the 99th and 100th percentile).

BBR starts to treat abrupt latency inflation as a congestion signal, reduces its windows rate, and increases PLT. We found that for 20% of the web page loads, DChannel performs worse than All-eMBB. These sites rely on a single TCP connection to deliver most web objects, and that flow suffers from a low sending rate. This can happen as DChannel accelerates the early packets (such as TCP SYN) to URLLC and suddenly switches back to eMBB once it sees large traffic. The abrupt RTT change gives a wrong congestion signal to the sender. One possible solution is to modify BBR to be aware of eMBB and URLLC use so that it can tolerate a change in RTT and maintain its sending rate. We leave this as future work. Alternatively, we can use different heuristics (pkt-uplink) that will send all uplink packets to URLLC and downlink packets to eMBB. This heuristic is based on the observation that client traffic is generally small and accelerating those packets will accelerate the flow RTT in a more consistent way. Table 4 shows we can get 39% improvement in PLT under BBR from this scheme.

3.4.5 Live 5G Experiments

We repeated the web-page fetches (similar to those in §3.4.3) over both the live-eMBB and emulated-eMBB settings. We then compared the relative improvements in PLTs

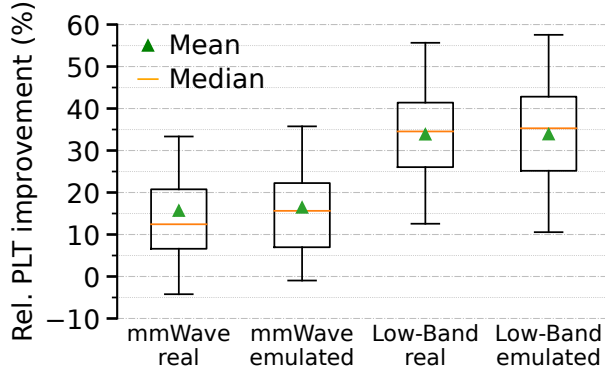


Figure 9: Comparison of relative PLT improvements in live-eMBB and emulated-eMBB settings⁷.

Table 5: The p50 and (p95) of the avg. and max. buffer sizes (in bytes) when loading 200 web pages under MM-S and LB-D traces.

	MM-S		LB-D	
	Proxy	UE	Proxy	UE
Avg buffer size (b)	2 (15.7)	12 (63.5)	14 (96)	130 (2638)
Max buffer size (b)	392 (1122)	944 (2597)	757 (2375)	2848 (15521)

brought about by DChannel across these settings, for both 5G mmWave and Low-Band (Fig. 9). In conclusion, the PLT improvements are quite similar between the live and emulated-eMBB settings, albeit the mean PLTs in the former are higher than those in the latter. The mean PLT for the All-eMBB scheme in the live-eMBB setting for 5G Low-Band is 1718 ms, while that in the emulated-eMBB setting is 1522 ms. The high mean PLTs in the emulated-eMBB settings could be due in part to the limitations of the setup. The emulations do not capture all eMBB network characteristics such as out-of-order delivery. Moreover, the web server is hosted on dedicated hardware in the live-eMBB setting, whereas in the emulated-eMBB setting it runs on the same laptop as the client. These minor setup differences, however, do not affect our claims concerning the relative performance improvements (which are similar across the two settings).

⁷The mean PLT improvements of mmWave is lower than what we reported on MM-S in Fig.4 because we used 8 ms of URLLC RTT (rather than 5 ms).

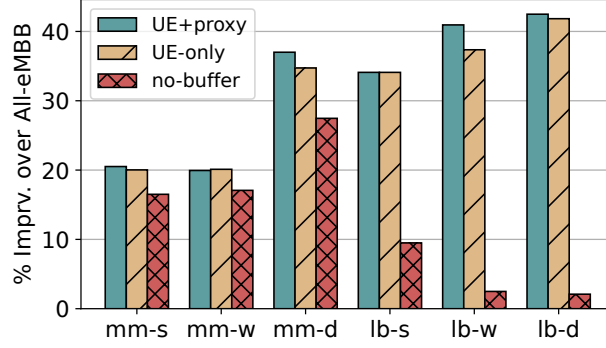


Figure 10: Adding reorder buffer to DChannel significantly improves web page load time.

3.4.6 Evaluating the reordering buffer

We evaluated the effectiveness of the reordering buffer (ROB) by comparing the web browsing performance of DChannel with and without the buffer. Along with the other experiments, this evaluation uses the default TCP CUBIC, which is sensitive to in-order packet delivery. Fig. 10 shows that without the buffer, the mean PLT improvement is decreased by up to 40% (in LB-D). Overall, DChannel without the buffer will be worse when the gap between eMBB and URLLC latency increases (implied in the figure as DChannel without the buffer is performing much worse in 5G Low-Band than mmWave) because DChannel will be more aggressive in offloading packets to URLLC, causing more out of order packets.

If ROB is implemented only on the UE (downlink), PLT improvement is only reduced by 2%. The decrease in PLT improvement is because DChannel tends to offload most web browsing uplink traffic to URLLC as the client requests are generally small such that minimal out-of-order persists. Our buffer analysis in Tab. 5 confirms that DChannel proxy does not use the buffer as frequently as the DChannel client (UE). Tab. 5 also shows that ROB requires only little memory as we do not have to buffer much URLLC traffic.

Table 6: PLT under different random packet drop rates.

Loss	All-eMBB (ms)		DChannel (ms)	
	MM-S	MM-D	MM-S	MM-D
0.0%	1108	1899	883 (20%)	1096 (42%)
0.1%	1203	1963	1011 (16%)	1311 (34%)
1.0%	2643	3421	2502 (5%)	3072 (10%)

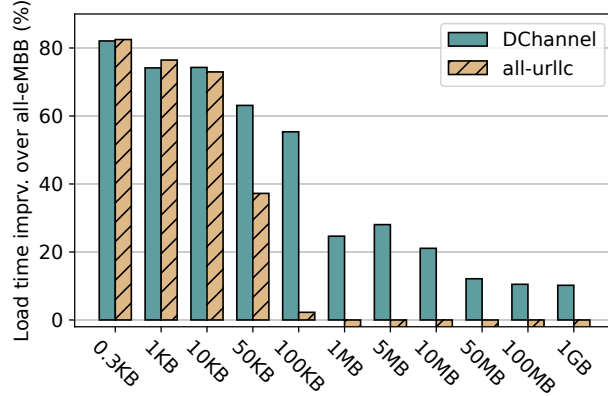


Figure 11: Download time improvement of variable-sized data under HTTP. The experiment used the MM-D trace with the buffer set to 800 packets ($\approx 2 \times$ trace BDP).

DChannel under random packet drop. Since ROB holds packets from URLLC for a fixed (100 ms) timeout when they are not in order, DChannel may suffer when packet loss happens as it may delay the packet loss signal, which can happen under certain conditions (§3.2.6). To evaluate this effect, we investigated DChannel performance under stationary and driving traces in the case of random packet drops. We applied a stochastic packet drop in the uplink and downlink channels with packets being dropped in both eMBB and URLLC. This reflects the case where packet drops happen on the Internet path. Note that 5G eMBB generally exhibits low packet loss rates with the 99th percentile being 1.2% [25]. Per Tab. 6, DChannel is quite resilient to random packet loss, offering better performance than All-eMBB even under high loss rates.

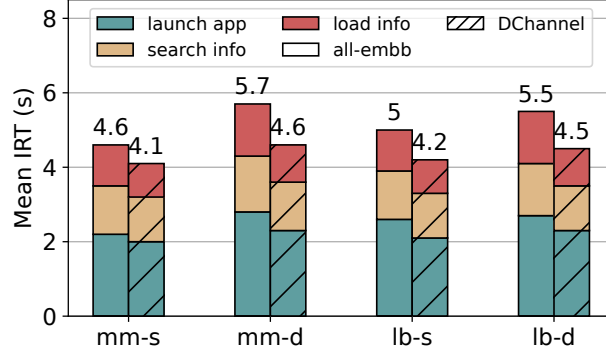


Figure 12: Android mobile application interaction response time (IRT) of All-eMBB and DChannel when performing three different tasks. We averaged the result from three applications.

3.4.7 DChannel rewards calculation accuracy

We evaluated DChannel packet rewards (R) calculation accuracy by comparing the calculated rewards and the *real* rewards. As we used a trace-based emulated network, we knew both network bandwidth, latency, and the link’s queue depth at any given time. Leveraging this information, we can calculate the *real* rewards before we commit a packet to a link. Figure 8 shows the rewards calculation error for the mmWave stationary (MM-S) and driving (MM-D) network traces. DChannel is able to accurately estimate R with just 12 ms of error in the 90th percentile in the stationary network traces due to the network latency being relatively stable and the bandwidth being large. In the driving traces, the error is noticeably higher with a long tail. However, 90% of the time the error is less than 37 ms. The R error mainly arises from the less accurate network eMBB latency estimation and the impact of ignoring the eMBB queueing delay (§3.4.4), since eMBB bandwidth may be low, and the delay becomes more significant.

The above is also why, as can be seen in the figure, R is rarely overestimated. Underestimation is the preferred direction of error, as we have shown that DChannel can tolerate some incorrect latency estimates and rewards underestimation (§3.4.4). However, better network measurement may improve DChannel performance; we leave this as future work.

3.4.8 Bulk download performance

Although our primary focus is latency-sensitive applications, we also evaluated how DChannel performed for a bandwidth-intensive use—bulk HTTP transfer of a file. Fig. 11 shows the download time for various file sizes using the mmWave driving (MM-D) trace. As expected, DChannel gets the largest improvement for small objects. But interestingly, DChannel also usefully improves large object download. This is because all control packets including TCP ACKs are accelerated in URLLC, which reduces the control loop delay, helping the transport layer adapt to bandwidth changes more quickly. Specifically, we found DChannel resulted in better utilization of the available eMBB throughput by $\approx 10\%$ when there are large throughput variations (e.g., in the driving scenario).

3.4.9 Mobile application performance

Fig. 12 shows the application response time improvement of DChannel across three common user interactions that require communication to the server. On average, DChannel improves the response times of application launch (15%), query searching (12%), and information (e.g., product or news) loading (21%). It is unsurprising that query searching gives lower improvement since it incurs higher server-side delay. The overall mobile apps improvement is lower than the web browsing in part to our experimental setup (§3.4.2).

3.5 Discussions on DChannel for 5G networks and beyond

Deployment for 5G networks. DChannel requires cellular operator support, to allow URLLC for non-critical traffic and to perform stateful packet steering. However, operators may omit some of the DChannel implementations to make deployment easier, such as eliminating the reordering buffer (ROB) on the core gateway since DChannel shows only minor performance degradation without ROB in the proxy-side (§3.4.6). DChannel stateful packet steering may not be simple to implement especially when there

are multiple gateways. We leave this to future work.

URLLC scalability. The number of users that can send general traffic to URLLC is an important matter which deserves to be evaluated quantitatively in the future. At the time of writing, URLLC is not yet deployed in public. However, based on the white paper [82], URLLC is targeted to support a relatively high connection density with modest per-user bandwidth. For instance, one of the URLLC use cases (discrete automation) requires a user-experienced data rate of 10 Mbps, traffic density of 1 Tbps/km², connection density of 100,000/km², and max end-to-end latency of 10ms. Thus, the 2 Mbps maximum bandwidth per user for general application traffic used in our experiments is still reasonable based on others' proposed use cases for URLLC, even in a dense urban area.

Disrupting URLLC native traffic. URLLC is primarily built to serve latency-sensitive critical applications. To ensure we do not compromise the performance of these applications, the network operator can limit the per-user bandwidth and even choose to deprioritize non-critical packets as our approach does not require 99.999% reliability and is resilient to small increases in URLLC latency (§3.4.4).

Resource contention among applications. Multiple applications inside a user device may compete to use URLLC. We can regulate them using prioritization. One simple approach is to prioritize applications running in the foreground since mobile phone users are typically single-tasking.

Incentives for operators. While URLLC targets critical applications, it is up to the network providers to open URLLC for general mobile applications like web browsing. This is possible as 5G chipsets are typically designed to support multiple bands including the sub-6GHz bands for URLLC [35]. Expanding URLLC applications can encourage

providers to foster a faster and broader deployment of URLLC as it brings a smoother experience to their major customers – mobile phone users; especially as the current market for URLLC applications like self-driving cars and remote surgery is still in its infancy.

Emulation uncertainty. The real URLLC performance might not match our emulated URLLC that follows the 5G NR white paper. However, we have performed several experiments to show the robustness of DChannel under variable URLLC conditions. Emulating the real behavior of a cellular network (eMBB) is also a known hard problem [83], and we specifically highlight and tackle this problem in the Chapter 5. We have compared DChannel performance with the emulated eMBB and live eMBB in stationary conditions and conclude that DChannel offers the same performance benefit (§3.4.5). However, we have not yet evaluated DChannel under non-stationary live eMBB due to the environment limitation (§3.3.1).

Other applications. LLC and HBC combination can also properly support applications from different domains that require high bandwidth and low latency, something that cannot be satisfied by utilizing a single channel. For instance, cloud gaming, which allows users to play games from remote servers, requires high bandwidth to stream content and low latency to remain responsive to user input. Since these applications can be vastly different than web browsing, a superior steering scheme may exist. For these type of applications, application inputs can also be critical to achieve a more optimal steering decision. We discuss this in more detail in Chapter 4.

Chapter 4 Informed application-transport service with DChannelX

4.1 The need to operate at the transport layer

We argue that the key to designing an effective packet steering mechanism (or scheduler) for high-bandwidth and low-latency channels is to effectively map application messages to the respective channel. As described in §3.2.3, a message is defined as a sequence of one or more packets such that the receiving endpoint can take some useful action only after receiving the full message. In addition, LLC, due to its limited bandwidth, should be used sparingly, ideally only for traffic that is latency-sensitive. Meanwhile, to obtain useful information about the messages, we have to operate at the transport layer and above.

4.1.1 The suboptimality of the network-layer approach

Although a network-layer solution like DChannel is easily deployable since it operates transparently without requiring modifications or input from the application, its scheduling decisions are inherently suboptimal because it lacks application-level information. Recall from §3.2.3 that our goal is to accelerate the delivery of messages, particularly those most valuable under the bandwidth constraints of the LLC. However, DChannel lacks two pieces of critical information: (1) message boundaries and (2) message importance or priority, neither of which can be inferred at the network layer.

Message boundaries can be very useful to provide a better estimate of the reward. DChannel uses packet completion time to estimate Rewards (R), treating every packet as a potential message boundary and thus optimistically considering it for acceleration. When sending a large message (indicated by a long burst of packets), DChannel will accelerate the first few packets to the LLC until the Cost (C) exceeds the Reward (R). Once this happens, the subsequent packets are sent to the HBC. This decision ultimately wastes LLC capacity, as those earlier accelerated packets cannot be used by the application until the full message completes. With explicit message boundaries, however,

Traces	eMBB-only	DChannel	DChannel w. priority
Stat.	1697.3	1230.5 (27.5%)	1154.9 (32%)
Drv.	2334.3	1474.6 (36.8%)	1336.8 (42.7%)

Table 7: Web PLT (in ms) with small background traffic using emulated 5G lowband eMBB (stationary and driving traces) with URLLC.

R can be estimated based on the full message completion time. In this example, we could send the entire large message over the HBC and reserve LLC capacity for shorter or more latency-critical messages.

Messages within an application can also have vastly different performance (or latency) requirements or priorities. For example, when loading a web page, some objects (e.g., JavaScript) are far more important than others (e.g., images). An effective scheduler should minimize completion time for these high-priority objects. By identifying which subset of application traffic is best suited for each channel and steering messages accordingly, we can deliver substantial benefits beyond what a purely network-layer solution like DChannel can achieve.

4.1.2 Potential benefits of application inputs

To highlight potential benefits of application input to steering, we now demonstrate the value of knowing even basic information such as flow priorities when multiple flows compete, through an experiment. We loaded web pages while also running two background flows that do not contribute to the PLT. These background flows upload and download small JSON files, a common occurrence for mobile web-based apps that upload logs to the server and prefetch data not yet displayed to the user. In one case, packets from all three flows are steered into eMBB or URLLC with DChannel. We then supplied DChannel with flow priorities to prevent the background flows from using the limited URLLC bandwidth.

Table 7 shows the web PLT for three different steering policies. Both versions of

DChannel improve PLT considerably over the eMBB-only case, where all traffic is delivered over eMBB. Nonetheless, DChannel with priority performs the best, reporting a 6.2% to 9.5% mean PLT improvement compared to DChannel without any flow prioritization. This improvement mainly stems from avoiding URLLC queue buildup caused by delivering low-priority background traffic over the channel. As a result, web page load traffic can be steered more toward URLLC and achieve greater acceleration.

In summary, this result presents significant benefits from incorporating minimal application-level information into a solution that leverages HBC and LLC.

4.1.3 Other downsides of the network-layer approach and how transport-layer solutions address them

In some cases, a network-layer approach like DChannel can interfere with transport-related logic because the steering is transparent to the transport. The most common examples are delay-dependent transport protocols like BBR [17], which rely on network measurements (i.e., round-trip time, or RTT) to perform actions such as adjusting the sending rate.

To demonstrate this, we ran tests on Pantheon [83] with DChannel used to steer packets between an HBC with 50 ms RTT and 60 Mbps bandwidth and an LLC with 5 ms RTT and 2 Mbps bandwidth. We tested three congestion control algorithms (CCAs) that adjust their sending rate based on packet delay – TCP BBR [17], TCP Vegas [84], and PCC Vivace [85] – with this setup. In addition, we tested TCP CUBIC [86], a loss-based CCA that is far less sensitive to delay. Figure 13a presents the results of these experiments. CUBIC, a loss-based CCA, utilizes the full throughput (60 Mbps) of the high-bandwidth channel. All delay-dependent CCAs underutilize the link: BBR, Vegas, and Vivace achieve average throughputs of 26.5 Mbps, 2.73 Mbps, and 1.49 Mbps, respectively.

We then plotted the measured RTT when running BBR over time in Figure 13b. The

RTT varies significantly, confirming our suspicions of transport-layer confusion caused by sudden changes in RTT. This is exacerbated by DChannel prioritizing control packets, including BBR’s probes, to the low-latency path. This produces an initial underestimate of the RTT, followed by an ostensible spike upon reverting to the high-latency path, leading to perceived congestion during the first 10 s of Figure 13b. At the 10 s mark, BBR drains the queue to obtain a better estimate of the minRTT. However, this estimate is lower than the true RTT of the high-bandwidth channel. As a result, BBR underestimates the Bandwidth-Delay Product (BDP) and is unable to fully utilize the high-bandwidth channel.

Building a transport-layer solution that is aware of both LLC and HBC and their respective network characteristics can eliminate these issues. Moreover, a transport-layer design enables additional optimizations that better utilize the limited LLC resources. For instance, ACK packets (as control packets) are highly beneficial to send over LLC because they are small and can trigger immediate actions at the sender (e.g., transmitting additional data). However, if an ACK is bundled with application data into a single packet, that packet is typically sent over HBC, since HBC offers higher bandwidth, resulting in a missed opportunity to accelerate the ACK via LLC. A transport-layer solution can easily decouple ACKs from data packets and send them over LLC.

4.2 DChannelX as the application-transport solution

We propose DChannelX, which offers a more robust solution to utilize the HBC and LLC more optimally. DChannelX operates at the transport layer, which eliminates any transport-level confusion that a network-layer approach can cause, and can also offer transport-specific optimizations such as decoupling ACK packets from data packets. The DChannelX network API can also accept useful information from the application and use that to help with steering. Compared to the network-layer solution, which requires no application intervention or modification, DChannelX requires applications to use this

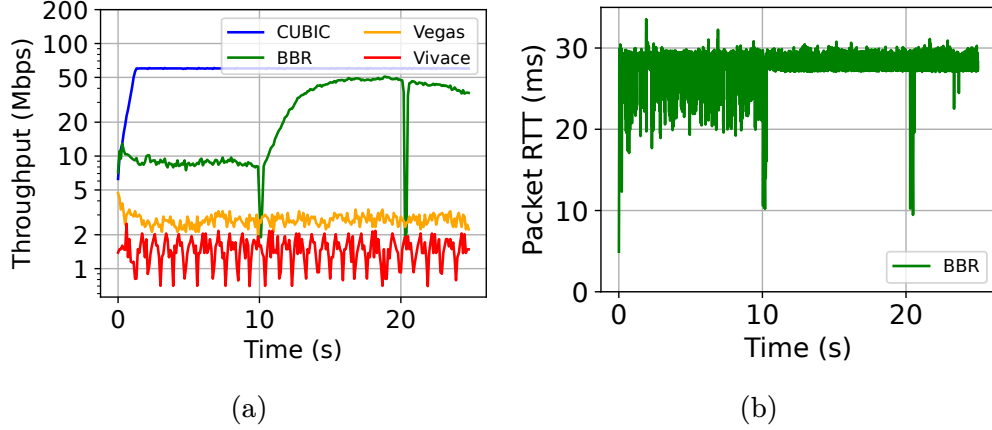


Figure 13: (a) Throughput achieved by CCAs with DChannel on two paths with a latency-bandwidth trade-off. Y-axis is in logarithmic scale. (b) Packet RTTs observed by BBR when using DChannel.

transport API and provide useful inputs.

4.2.1 High-level design

DChannelX is implemented as a packet scheduler built on top of an existing multipath transport protocol. We use Multipath QUIC (MPQUIC) rather than Multipath TCP (MPTCP) because QUIC’s user-space implementation allows richer and more flexible modifications. DChannelX requires the ability to forward ACKs from one path (e.g., HBC) to another path (e.g., LLC), which MPTCP does not support. Applications interact with DChannelX through a message-based network API, which internally invokes MPQUIC’s stream-based API. This interface allows applications to send data (i.e., messages) and optionally provide metadata, such as message size and priority, to guide traffic steering decisions. The API maintains a one-to-one mapping between a message and a QUIC stream, meaning that each message is assigned a unique stream (i.e., a distinct stream ID).

As described in §3.2.2, effective utilization of the LLC requires fine-grained steering. DChannelX steers traffic at the granularity of packets while maintaining awareness of each packet’s association with its corresponding message. Messages can be either

application-level (e.g., DNS queries or HTTP requests/responses in web browsing) or transport-level (e.g., ACKs). At a minimum, the application must provide message boundaries through the API for DChannelX to operate correctly. Optionally, it can also specify message priorities to further assist the steering decision.

DChannelX centers around a packet scheduling algorithm that extends MPQUIC’s scheduling framework. In MPQUIC, packets are scheduled whenever one or more paths are available, as governed by the congestion control mechanism (i.e., when permitted by the congestion window, or *cwnd*). Each time a path becomes available, DChannelX examines the data (or stream) queue to determine the most suitable path for each pending packet. If a packet is selected for transmission on an unavailable path, it is temporarily queued until that path becomes ready.

4.2.2 Steering design and algorithm

DChannelX follows the rewards-and-cost approach from DChannel (§3.2.4). Below, we describe how DChannelX computes its rewards and cost functions.

Rewards. Our goal is to minimize the message completion time T_M , defined as the estimated time at which all packets belonging to a message M have arrived at the receiver. The benefit of sending a message M over LLC is therefore the reduction in T_M when all packets of M are sent via LLC (denoted T_M^{LLC}) compared to when they are sent via HBC (denoted T_M^{HBC}).

Unlike common MPTCP or MPQUIC scheduling algorithms that may distribute packets of a single message across multiple paths to achieve bandwidth aggregation, DChannelX currently does not split packets of a single message across channels. Such splitting is only beneficial for large or medium-sized messages, where sending the entire message over LLC would take longer than over HBC. However, since LLC typically provides only a small fraction of the bandwidth available to HBC (e.g., $\approx 1\%$ in 5G

eMBB and URLLC), offloading packets from those messages to LLC would yield only negligible improvements in completion time. Thus, we reserve LLC capacity only for smaller, latency-sensitive messages (such as transport-layer ACKs).

We calculate the rewards of sending M via LLC as:

$$R(M) = T_M^{\text{HBC}} - T_M^{\text{LLC}} \quad (6)$$

The completion time T_M^{link} of a message M over a given link depends on the link propagation delay ($D_{\text{prop,link}}$), bandwidth (B_{link}), message size ($\text{size}(M)$), and the instantaneous queue size at time t_n ($Q_{\text{link}}(t_n)$). Here, Q_{link} represents the total number of bytes that have been enqueued for transmission on that link plus all bytes that have already been assigned for that link's upcoming transmission.

The completion time of M is thus:

$$T_M^{\text{link}} = D_{\text{prop,link}} + (\text{size}(M) + Q_{\text{link}}(t_n))/B_{\text{link}} \quad (7)$$

Cost. As in §3.2.4, the cost of sending a message to LLC reflects the additional utilization of the channel, specifically, the added queuing delay that a subsequent message arriving shortly after M would experience.

We define the cost (or fare, F) of sending M via LLC as:

$$F(M) = (\text{size}(M) + Q_{\text{llc}}(t_n))/B_{\text{llc}} \quad (8)$$

Comparing rewards and cost. DChannelX computes the rewards and cost when processing the first packet of each message. All packets belonging to message M are sent over LLC if:

$$R(M) > \alpha F(M) \quad (9)$$

The parameter α controls how aggressively DChannelX utilizes the LLC. A higher α increases the effective cost of using LLC, thus requiring greater estimated reward for a message to qualify for transmission on the low-latency channel. DChannelX ties this parameter to the message’s priority as provided by the application, in which higher-priority messages use smaller α values. If no message priority information is available, α defaults to 0.75, which is similar to DChannel.

Embracing Priority. To enable priority-based steering, applications need to specify a range of priority values. The parameter α is then assigned to each priority value P using the following linear function:

$$\alpha = \alpha_{min} + P * (\alpha_{max} - \alpha_{min}) / (P_{max} - P_{min}) \quad (10)$$

Here, P_{min} and P_{max} denote the minimum and maximum priority values specified by the applications. Unless stated otherwise, we use $\alpha_{min} = 0.5$ and $\alpha_{max} = 5$ in our experiments.

In addition to controlling LLC usage through α , DChannelX also leverages the built-in priority-based stream (or message) scheduling mechanism in QUIC, where higher-priority streams are prioritized to be sent before lower-priority ones.

Algorithm 2: DChannelX steering algorithm

Result: Send a message to either HBC or LLC
 $T_{llc} = llc_prop_delay + (size(message) + queue_llc) / bandwidth_llc;$
 $T_{hbc} = hbc_prop_delay;$
 $rewards = T_{hbc} - T_{llc};$
 $cost = (size(message) + queue_llc) / bandwidth_llc;$
if $rewards > \alpha * cost$ **then**
 | send(message, LLC);
else
 | send(message, HBC);
end

The steering algorithm The complete steering algorithm is presented in Algorithm 2. To make a scheduling decision, the algorithm requires four inputs: (1) the message size, (2) the current LLC queue size, (3) the LLC bandwidth, and (4) the latency (i.e., propagation delay) of both LLC and HBC. As in DChannel, we omit the HBC queue delay when computing R to simplify the problem, since HBC typically offers much higher bandwidth than LLC and its bandwidth is harder to estimate in real time. However, because DChannelX is integrated with MPQUIC, it may be possible to leverage the HBC bandwidth estimation that is already available from the transport layer (e.g., as reported by the congestion controller). We leave this integration as future work.

The message size (1) is provided by the application. The LLC queue size (2) is obtained using the same method described in §3.2.5. For latency (4), transport-level measurements can be used, particularly those collected during MPQUIC’s connection establishment and path discovery phases.

4.3 DChannelX implementation

We implemented DChannelX as a custom MPQUIC packet scheduler using Picoquic as the underlying QUIC implementation. In MPQUIC, a single connection can have multiple active paths. Our scheduler’s primary role is to steer or assign packets to the HBC or LLC paths according to the DChannelX steering algorithm.

We provide a message-based network API that internally invokes Picoquic’s `picoquic_add_to_stream` function to send application messages. The API accepts message size (required) and message priority (optional) as application inputs. Internally, `picoquic_add_to_stream` inserts data into Picoquic’s stream tree and eventually triggers the function `picoquic_prepare_packet_ex`, which in turn calls `picoquic_select_next_path` to choose an available path (e.g., based on congestion control or `cwnd`) and then calls `picoquic_prepare_segment` to retrieve data from the stream tree and construct a QUIC packet for transmission.

The DChannelX steering algorithm is implemented within `picoquic_select_next_path`. To make a path selection decision, the algorithm requires information into the next packet that will be formed by `picoquic_prepare_segment`, such as whether it will be a small ACK packet or data packet, and which stream (message) it belongs to. Rewards and cost calculations are performed whenever the scheduler detects that the next packet belongs to an unassigned stream (i.e., the first packet of a message without a designated path). Once the algorithm determines the appropriate path, the stream is marked with a corresponding Path ID, and all subsequent packets from that stream (message) are sent through the assigned path.

We also modify `picoquic_prepare_segment` to ensure that each packet carries data from only a single stream and that ACK packets are not combined with data packets within the same QUIC packet. This modification allows a one-to-many mapping between messages (either application-level or transport-level) and QUIC packets, ensuring clear message-to-path correspondence. Finally, we still use Picoquic’s built-in receiving API.

4.4 Preliminary evaluation

4.4.1 Experimental setup

Similar to DChannel evaluation, we use a client–server setup representing a mobile end-user application fetching content from a remote content server. Both the client and server endpoints have access to two emulated network interfaces—corresponding to HBC and LLC—and communicate using the DChannelX API. We use 5G eMBB and URLLC as the reference configurations for HBC and LLC, respectively. eMBB has 60ms RTT and 180 Mbps of upload and download bandwidth, while URLLC has 5ms RTT and 2 Mbps of upload and download bandwidth. The two interfaces are emulated using `mpshell` [87]. Unless stated otherwise, both the client and server run on the same laptop equipped with 16 GB RAM, a 512 GB SSD, and an Intel Core i7 processor running Ubuntu 20.04 (Focal Fossa).

4.4.2 Applications under test

We evaluated DChannelX using a client–server setup that simulates web browsing traffic. Unlike DChannel, which can steer traffic without modifying the application, DChannelX requires applications to use its API. This implies that real-world web browsers (e.g., Chrome) and web servers (e.g., Apache) would need to be modified to interface with Picoquic and the DChannelX API. However, to the best of our knowledge, no existing web browser or server currently supports Picoquic. Therefore, we developed WebSim, a lightweight web browsing client and server simulator written in C++ that use Picoquic and send traffic through the DChannelX API.

Web browsing simulation. WebSim simulates web browsing traffic by traversing the page dependency graph [69] captured using WProfX [88]. WProfX instruments the Chrome browser to record page load activities and infer the dependency graph from those observations. We further modified WProfX to capture the priority levels assigned by Chrome’s internal prioritization mechanism [81], which classifies web resources into five static levels: **VeryHigh**, **High**, **Medium**, **Low**, and **VeryLow**. These priorities are assigned based on the resource type and its location within the root HTML file.

We then replayed the recorded dependency graph in WebSim, following an approach similar to EpLoad [68]. Specifically, WebSim walks through the graph starting from the initial activity—the loading of the root HTML. When it encounters a network activity, the client sends an HTTP request to the corresponding server URL and waits for the HTTP response. When it encounters a computation activity, it sleeps for the duration indicated by the dependency graph. After completing each activity, the client checks whether dependent activities can now be triggered—that is, whether all their prerequisite activities have completed. This process continues until all activities in the dependency graph have been visited. The *Page Load Time* (PLT) is then defined as the total time required for all activities in the dependency graph to complete.

Table 8: Distribution of Chrome-assigned priorities for web objects across the evaluated pages.

Priority	#Object (%)	Mean obj size
VeryHigh	557 (14.8%)	15,813 bytes
High	376 (9.99%)	10,169 bytes
Medium	223 (5.92%)	13,712 bytes
Low	2608 (69.29%)	15,996 bytes
VeryLow	0 (0%)	0 bytes

The WebSim client is a multi-threaded program with a configurable number of threads, each capable of processing an activity (either waiting for an HTTP response or simulating a computation delay). HTTP requests are issued through the DChannelX API, which also reports the message size corresponding to the total HTTP request data size. WebSim also supports multiple concurrent connections to the server, where each unique URL hostname maps to a distinct connection ID. Due to a high connection setup delay we observed when establishing MPQUIC connections with two paths in Picoquic, we decided to pre-establish all connections to ensure both paths are available for every single connection prior to replaying the dependency graph.

4.4.3 Methodology

We used WProfX to load real web pages and record their dependency graphs on the same laptop described earlier, connected to a home Wi-Fi network. Our experiments use the web page corpus introduced in §3.4.2, from which we randomly select 54 pages belonging to the landing pages category. Table 8 summarizes the distribution of web objects by the priority levels assigned by Chrome. Across all evaluated pages, no objects are labeled as **VeryLow**. The recorded dependency graphs are then fed into WebSim, which replays them over our emulated network interfaces. Each page load is repeated ten times, and we reported the average (mean) Page Load Time (PLT) across runs.

4.4.4 Object download time improvement

We compared DChannelX against two baselines: (1) the default MPQUIC packet scheduler in Picoquic (referred to as Picoquic-MP) and (2) our implementation of DChannel packet steering on top of MPQUIC. We also include a single-path QUIC as the baseline (all-eMBB), where all traffic is sent exclusively over the eMBB path.

Picoquic-MP selects path based on the lowest measured round-trip time (similar to the min-RTT strategy used in MPTCP’s default scheduler). The DChannel MPQUIC implementation follows the algorithm from §3.2.5, steering individual packets without awareness of message boundaries. We evaluated two variants of DChannelX: (1) DChannelX-msg, which receives only the message boundary information from the application, and (2) DChannelX-msg-priority, which additionally incorporates application-provided message priorities. For DChannelX-msg-priority, we define four priority values (Table 8) with α values set to 0.5, 2, 3.5, and 5, corresponding to decreasing priority.

Figure 14 presents the object download time improvement (grouped by Chrome’s priority classes [81]) relative to the all-eMBB baseline. The default Picoquic multipath scheduler performs worse than the single-path baseline by an average of -81.55% across all priorities when operating over eMBB and URLLC. This is consistent with our previous findings on the default MPTCP scheduler. In contrast, DChannel improves the average object download time by 12.59%. Note, however, that this improvement is lower than the gains reported earlier in §3.4, since DChannel’s primary benefit arises during the connection setup phase (by accelerating control packets such as SYN and SYN/ACK over URLLC). Since all connections were pre-established in this experiment, DChannel could not leverage its advantage during the connection setup phase. Using application-provided message boundary information (e.g., HTTP request and response sizes), DChannelX further improves the overall download time by 24.52%.

When priority information is included in DChannelX-msg-priority, the scheduler

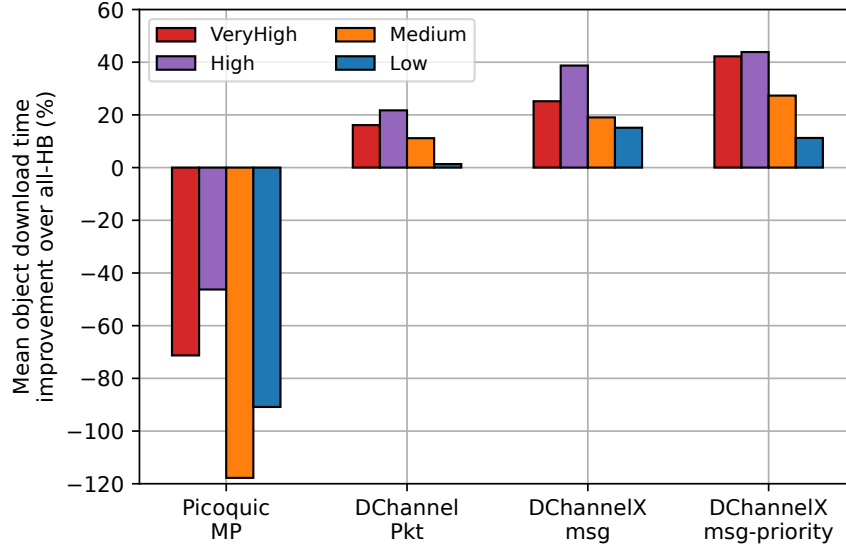


Figure 14: The web object download time improvement compared to the baseline (all-eMBB) grouped by its priority.

achieves additional improvements for higher priority objects. Specifically, **VeryHigh**, **High**, and **Medium** objects show added improvement in download time of 17.04%, 5.12%, and 8.28%, respectively, compared to DChannelX-msg. As expected, **Low**-priority objects experience a slight degradation (-3.88%) because their α value is set to 5, making it harder for the reward function to exceed the scaled cost threshold. Consequently, only smaller **Low**-priority objects are sent over URLLC. These results demonstrate that adjusting α according to application-defined priorities can be an effective way to honor priorities given by the application. However, determining the most effective priority assignment to optimize higher-level metrics (e.g., overall page load time) remains a challenging open problem, as we demonstrated in the next section.

4.4.5 Page load time improvement

Next, we focus on the web page load time (PLT) improvement. Interestingly, we found that DChannelX with priority information from Chrome does not help in reducing web PLT when compared to DChannelX without priority information. Upon closer examination, we observed that some objects marked as “low” priority actually lie on the

critical path of the page load process. The critical path (CP) represents the sequence of activities that forms the longest path in the dependency graph. The page load time can only be reduced by reducing the duration of any activity on this path [88]. With DChannelX, Low-priority objects (which may lie on the critical path) experience increased loading delay because they are less likely to use URLLC, and this offsets the gains from faster loading of other objects. This observation motivates the use of a different priority assignment strategy – one that directly bounds the structure of the page’s dependency graph through critical path analysis.

Assigning object priorities via critical path analysis. To perform a critical path analysis, consider a dependency graph $G = (V, E)$, where V (nodes) represents the set of activities required to render the page, and E (edges) represents the dependencies among them. Each activity $a \in V$ is associated with a duration indicating its execution time. Let P denote the set of all paths from the root node to any leaf node in G . The page load completes once every path in P has been traversed. The critical path CP ($CP \in P$) is defined as the path with the longest cumulative duration, such that reducing the duration of any activity not on this path does not affect the total PLT.

However, identifying only the single longest path is insufficient, as optimizing the load time of objects on CP can alter the graph’s dependency structure and potentially shift the critical path. To account for this, we define P_{thresh} as the subset of paths whose traversal durations exceed a threshold $thresh$. In our analysis, we set $thresh = 0.5 \times \text{duration}(CP)$, i.e., half the duration of the critical path. Thus, for DChannelX, we use 2 priority levels, and set every objects that belongs to P_{thresh} to priority level 0 and the others to 1.

Result. Table 9 summarizes the web PLT improvement relative to the baseline (all-eMBB). The default Picoquic-MP scheduler performs worse than the single-path baseline, with an average degradation of 30.58%. DChannel improves PLT by 11.01%,

Table 9: Web page load time (PLT) under the baseline (single path QUIC) and various multipath scheduling schemes.

Scheme	Mean PLT (ms)	% Improvement over all-eMBB
all-eMBB	623.34	-
Picoquic-MP	807.62	-30.58%
DChannel	560.67	11.01%
DChannelX-msg	493.8	21.84%
DChannelX-msg-prio	474.04	24.19%

while DChannelX —with message boundary information—further outperforms DChannel, achieving a 21.84% improvement over the baseline. When incorporating object priorities derived from critical path analysis, DChannelX achieves an additional 2.35% gain, resulting in an overall improvement of 24.19% compared to the baseline.

Table 9 shows the web PLT improvement compared to the baseline (all-eMBB). Picoquic-MP unsurprisingly performs worse than the single path baseline (by -30.58%). DChannel manages to improve PLT by 11.01%. DChannelX with message boundaries information manages to beat DChannel and improve the PLT by 21.84% compared to the baseline. With the object priorities derived from the critical path analysis, DChannelX manages to get 2.35% additional improvement which leads to an overall improvement of 24.19%.

4.5 Discussion on DChannelX and future work

More dynamic steering algorithm. DChannelX determines the path assignment for a message when the first packet of that message is about to be sent, and this assignment remains fixed for all packets of that message. We plan to design a more dynamic steering algorithm that may change the a packet’s path assignment in response to certain conditions, such as the arrival of higher-priority messages or when the remaining bytes of a message become small.

This dynamic approach could improve performance, as the path assignment for a

message may no longer be optimal as conditions change. In such a design, the steering decision would be made each time a packet is about to be sent. However, depending on the algorithm, this approach may introduce additional processing overhead, since the scheduling algorithm would need to run for each packet instead of for each message.

Richer application inputs. Currently, DChannelX supports a mandatory message size and an optional message priority as inputs. We plan to expand DChannelX API to accept additional application-level hints that can further assist traffic steering. Potential inputs include: (1) Message deadline, that allows application to specify hard or soft delivery deadline for each message. (2) Message drop tolerance, to indicate whether a message can be dropped if higher-priority messages compete for the low latency channel. (3) Message ordering constraints, to specify whether messages should be delivered in a particular sequence.

DChannelX for other applications. DChannelX is designed as a generic scheduler that can be applied across diverse applications. In this thesis, we primarily evaluated it using web browsing traffic because it presents a complex workload with varying message priorities, making it a representative and challenging case study. In future work, we plan to extend and evaluate DChannelX in other domains such as XR, cloud gaming, and real-time video streaming.

In the next step, we also plan to further tightly integrate the application and the transport to enable more optimal scheduling of traffic between the low-latency and high-bandwidth channels. Since different applications exhibit distinct traffic patterns, DChannelX’s steering algorithm can be tailored to suit each profile. For instance, XR workloads often exhibit quasi-periodic traffic patterns [15], which can be leveraged to design more effective steering algorithms.

Chapter 5 Towards an accurate record-and-replay for cellular network emulation with CellReplay

Cellular network performance, including bandwidth and latency, can vary significantly due to factors such as wireless interference, environmental obstructions, and handovers, especially in mobile environments [19], [89], [90], [91], [92]. The gold standard for evaluating application and protocol performance on cellular networks is, hence, to test them directly on *live* cellular networks. However, live testing is time-consuming, as experiments must be conducted across many different network conditions and can produce different results—due to different signal strengths, types of wireless service (e.g., 5G millimeter wave, 5G low-band, and 4G), kinds of interference, rates of mobility, physical locations, etc. Repeating each experiment multiple times is crucial to ensure statistically reliable results given the performance variability in cellular networks. In addition to being time-consuming, experiments are often difficult to reproduce. A lack of control over the environment makes it infeasible, for instance, to compare the effects of a protocol change under identical network conditions.

Thus, researchers and app developers often turn to simulation or emulation for much of their evaluation, hoping to replicate a representative environment that yields performance similar to a live network. Simulators and emulators, such as `ns3` [93] or Linux’s `netem-tc`, offer various options for configuring delay, jitter, bandwidth, packet loss, and more. While they can adjust these parameters to emulate specific and realistic conditions, properly tuning them to accurately represent the dynamic behavior of real-world cellular networks remains a challenging and open problem.

A more realistic approach is to *record* network performance traces (e.g., latency, bandwidth, or packet loss) over time using predefined workloads (e.g., RTT probing) on a real-world network and *replay* those traces in an emulated network for the tested apps. This method allows for recording different traces under various conditions (e.g., locations) and testing multiple apps using such recorded traces. Record-and-replay

emulation was pioneered by Noble et al. [94]. More recently, the Mahimahi network emulator [72] can also replay recorded cellular network traces and has been instrumental in the design and evaluation of several notable networked systems and protocols (e.g., [95], [96], [97], [98], [99], [100], [101], [102], [103], [104], [105], [106], [107], [108], [109]).

However, we found that Mahimahi can produce inaccurate results compared to real-world tests in important cases, particularly for latency-sensitive and bursty workloads. For instance, in our evaluation, we observed an average bias of approximately 17.1% in web page load times (PLTs) when comparing Mahimahi emulation to running the application in the same commercial cellular environment where the traces were recorded. This error is a persistent underestimation of the PLT rather than just random variation. This issue affects other applications as well and the error may even be greater. For example, we observed a 49% error for 250 KB file downloads when Mahimahi emulated a commercial Verizon 5G as shown in §5.4.6.

Thus, despite record-and-replay emulation being practical and widely used, it does not support high-fidelity testing of networked systems and protocols. Minimizing emulation error is crucial, particularly for wireless protocol and application research, where record-and-replay emulations are often the most feasible evaluation platform. These errors could affect any evaluation and may even alter its conclusions, as we demonstrated in the ABR algorithms use case (§5.4.10). Therefore, we asked: What is causing this emulation error? And, is there a way to fix it to faithfully record and replay real-world cellular network performance?

Our first contribution is to study how the record-and-replay method used by Mahimahi can result in persistent bias (§5.2). Mahimahi records *packet delivery opportunities* by continuously saturating the link with packets (a “saturator” workload) and noting when packets arrive at the endpoint. It then replays this trace as a schedule for when the link can deliver packets after delaying those packets using a *fixed* propagation delay, for any workload.

However, we found that this method causes two fundamental issues. First, it fails to fully capture network base latency changes, which are prevalent in cellular networks. In fact, our measurements show that Mahimahi underestimates RTT by 13.25% and 16.88% across two operators. Second, the available bandwidth that a cellular network provides to an end-to-end connection depends significantly on that connection’s workload⁸. For example, in our measurement using Verizon 5G, a long train with 100 back-to-back packets experiences 2.6 times higher delivery rate than a short train with 10 packets. In such cases, Mahimahi’s saturator (i.e., heavy traffic) would see a higher rate than what shorter traffic should experience. This dependency between cellular network available bandwidth and workload poses a fundamental challenge for record-and-replay because the whole point is to record *one* trace (which is necessarily running one workload) and replay that trace under a *variety* of applications for testing. If available bandwidth depends on the workload, is faithful record-and-replay feasible?

Our second contribution is to address these fundamental problems in a record-and-replay system called *CellReplay*. To solve the workload-dependence problem, one obvious approach would be to record performance under every possible workload. However, this is impractical and degenerates into simply testing every application directly on the live network, which is what record-and-replay emulation is trying to avoid. In other words, we can only record a *limited* number of different workloads. Another option would be to build a white-box emulation of providers’ underlying resource allocation policies; but these are proprietary and vary across providers, so we seek a black-box method based on end-to-end observations.

The approach we take is to record just two representative workloads (*light* and *heavy*) simultaneously, chosen at extremes on the range of traffic patterns, and then interpolate

⁸To be clear, this effect is not due to queuing behavior of a traditional link with constant throughput and latency; nor is it caused by variations in wireless physical channel quality over time. Although commercial providers’ internal policies are proprietary and opaque to us, the effect could be explained by a resource allocation policy (e.g. [110]) or a carrier aggregation policy (as observed in [111]) at the Radio Access Network (RAN) allocating some bandwidth for the client, but observing the client’s injected packets and dynamically modifying that provided bandwidth.

between them during replay to achieve high accuracy across a wide range of workloads. During the recording phase, we use two phones: one running a heavy saturator workload and the other running a light workload. The light workload is calibrated to capture RTTs and light-workload bandwidth, but is not too light as to capture the network’s transition from light to heavy bandwidth allocations. During replay, the emulator applies delay using the RTT trace and initially releases packets according to the light trace. It then splices in the heavy trace during longer packet sequences before eventually returning to the light trace after an idle period. This technique addresses the two key problems with Mahimahi’s approach mentioned above, namely capturing (1) dynamic RTTs and (2) bandwidth that depends on workload.

We implemented CellReplay using an architecture similar to Mahimahi—an emulated network interface that can be used by unmodified applications. Using randomized trials, we evaluated CellReplay’s accuracy by comparing the application performance when running under CellReplay emulation to the live networks. We tested two commercial providers’ 5G mid-band and low-band deployments, and covered multiple network conditions, including non-ideal conditions (e.g., in a crowded library) and mobility (e.g., driving). We evaluated two real-world application traffic patterns: randomized file downloads and web page loads with HTTP/1.1 and HTTP/2. These applications cover a variety of workloads, ranging from periodic small to heavy flows in file downloads to complex interleaved traffic from web page loads. Additionally, we used CellReplay to evaluate the startup phase of multiple adaptive bitrate (ABR) implementations for 4K video streaming.

We find that CellReplay substantially reduces emulation error. In web page load tests, CellReplay reduces emulation error from 17.1% with Mahimahi to 6.7%, representing a 60.8% improvement. For randomized file download tests, CellReplay lowers mean file download time errors from 7.9%-49% with Mahimahi to just 0.2%-22.4%. Moreover, CellReplay achieves lower error when replicating application

performance under non-ideal network conditions, such as inside a basement (15.22% error in Mahimahi vs. 5.87% in CellReplay) and a crowded library (22.51% vs. 8.47%), and during user mobility, such as walking (14.48% vs. 4.13%) and driving (13.15% vs. 6.97%). Finally, we demonstrate CellReplay’s usefulness in evaluating ABR algorithms, as it preserves the relative ordering of ABR performance and avoids the biases observed in Mahimahi. We discuss challenges and future directions for improvement in §5.5. We release CellReplay alongside with its recorded traces as an open source at <https://github.com/williamsentosa95/cellreplay>

5.1 Background and related work

5.1.1 Cellular network record-and-replay

The goal of record-and-replay network emulation (within our scope) is to emulate the end-to-end network performance of an application communicating between two endpoints, ensuring performance similar to that of a live network counterpart. During the recording phase, user equipment (UE) and the server send traffic according to a predefined workload (e.g., sending packets beyond the link bottleneck rate), while observed performance metrics (e.g., throughput) are logged. This workload should be *independent* of the tested applications, allowing us to record traces *once* and reuse them for multiple applications, regardless of whether they are UDP- or TCP-based. During replay, this trace is consumed by an emulated network interface. Real applications (e.g., a web browser and web server) can connect through this interface, and traffic between the endpoints will experience artificial network conditions (e.g., time-varying latency and bandwidth) as if they were communicating over a cellular network, even though they reside on the same physical host. Our goal is for any metrics of interest—including transport-level and application-level metrics such as flow completion time or web PLT—to closely match those of the live network.

Record-and-replay can be applied to any type of network, but our interest here is on

cellular networks. Their performance can be time-varying, vendor-dependent, and environment-dependent, making it difficult to generate conditions—whether in simulators, emulators with hand-picked or even calibrated [112] parameters, or testbeds—that match real-world complexity. Thus, record-and-replay is especially useful in such environments, but it is also challenging to execute well.

Note that record-and-replay deals with end-to-end conditions and does not require any link- or physical-layer information or support from network operators. Like past work, we do not need to determine which hops along the path cause certain performance effects. This means that the observed performance, and its replay, may result from a combination of sources (e.g., the 5G RAN, service provider core, or the Internet to a remote endpoint). However, major performance variations are expected to originate from the cellular network [89]. We sometimes refer to the observed performance as coming from a cellular link, the path, or simply the network; all terms are equivalent for our purposes.

5.1.2 Related work

Network emulators. Popular network emulators, such as NetEm [113] and dummyNet [114], can emulate cellular networks. Google Chrome also provides configuration profiles with fixed latency and bandwidth for cellular networks, such as "Fast" and "Slow" 3G [115]. Pantheon [112] provides calibrated emulators based on parameters like fixed propagation delay, bottleneck link rate, isochronicity, etc. These configurations are tuned to match packet traces collected from a path (including cellular network) using various congestion control protocols. iBox [116] extends this by incorporating cross-traffic. However, fixed parameters, by definition, do not capture time-varying effects, which are common in cellular networks.

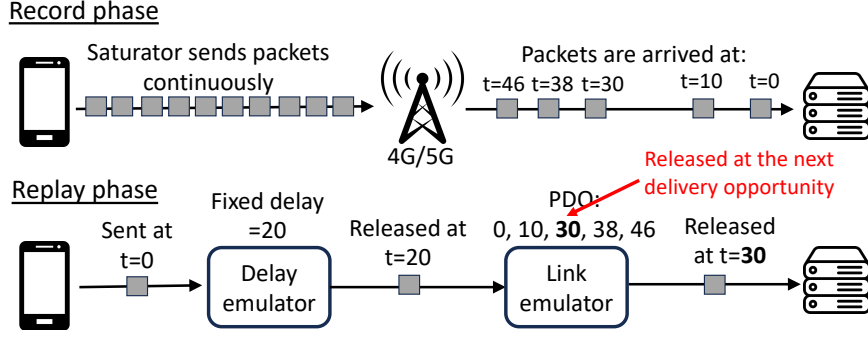


Figure 15: An overview of Mahimahi’s record-and-replay to emulate cellular uplink.

Record-and-replay network emulation. Noble et al. pioneered the concept of recording the end-to-end network characteristics of a wireless network and replaying them in an emulated network in 1997 [94]. However, it is designed to emulate WaveLAN, which differs fundamentally from modern cellular networks. More recently, NemFi [117] was introduced as a record-and-replay emulator for WiFi. NemFi’s design is specific to WiFi (e.g., emulating frame aggregation) and it is not readily applicable to emulating cellular network paths.

Mahimahi. In 2015, Netravali et al. demonstrated a framework for recording and replaying HTTP traffic [72] called Mahimahi, which also included a network emulator derived from CellSim [18] to replay time-varying uplink and downlink rates in cellular networks. Mahimahi has since become the state-of-the-art record-and-replay emulator for cellular networks and is widely used to evaluate various networked applications.

We detail Mahimahi’s record-and-replay approach⁹, as it serves as an important reference for this paper. Fig. 15 illustrates the process for the uplink only, as the same approach applies to the downlink. Mahimahi records time-varying link rates using a *Saturator*, which saturates both the uplink and downlink with MTU-sized packets (e.g., 1500 bytes) to ensure that the base station always has packets to deliver. The endpoint

⁹This method was introduced by CellSim. Mahimahi also provides traces recorded in CellSim’s approach, which have been beneficial and used in past work (e.g., [95]). Due to its usefulness, other work has collected newer cellular network traces following CellSim’s approach (e.g., [57], [109]) and replayed them in Mahimahi. For simplicity, throughout the rest of this paper, we refer to this record-and-replay method as Mahimahi.

then records the arrival time of each packet. During emulation, Mahimahi treats each arrival timestamp as an opportunity to deliver a packet. A sequence of such timestamps constitutes a *packet delivery opportunity (PDO) trace*. Each PDO entry represents an opportunity to deliver an MTU-sized amount of data, which can be either a single MTU-sized packet or multiple smaller packets whose combined sizes add up to the MTU. If no packets are queued for delivery when the PDO occurs, the opportunity is lost. Mahimahi also emulates the RTT delays on a cellular link, albeit using a *fixed* propagation delay. That delay is determined by measuring the minimum packet RTT (e.g., via ICMP ping) and halving that value.

5.2 Live record-and-replay is hard

Why is record-and-replay challenging? Also, why does the current state-of-the-art method (i.e., Mahimahi) fail to accurately replicate the performance of networked applications on a cellular network? Below, we answer both questions using measurements and insights from real cellular networks.

The measurements in this section were collected from two commercial cellular networks: T-Mobile 5G mid-band and Verizon 5G low-band, using a Samsung Galaxy S22 (SGS) phone tethered to a laptop. The laptop, equipped with an Intel i7 CPU and 16 GB RAM, ran Ubuntu 20.04 and served as our client. The server was located within close proximity (<10 miles) of the client. We confirmed that all results remained consistent when using a different phone model (Google Pixel 5).

5.2.1 Variability in base RTT

The base RTT is defined as the round-trip time (RTT) of a packet from the client to the server when there is no self-inflicted congestion. In cellular networks, this RTT is expected to be variable, as packets frequently experience delays (jitter) due to link-layer retransmissions, channel contention, base station scheduling, and device mobility.

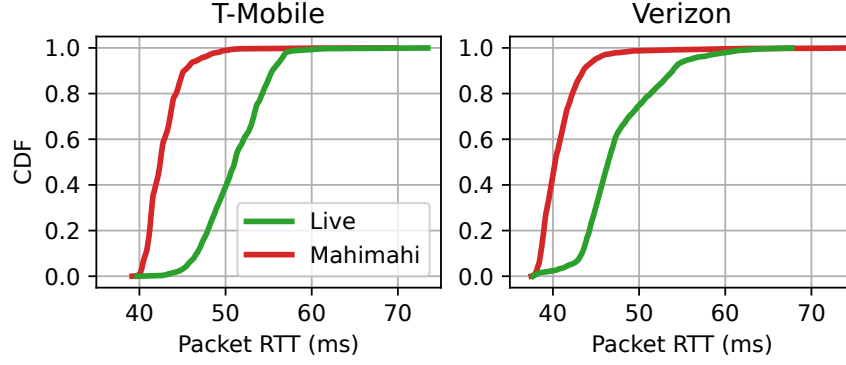


Figure 16: Packet RTT CDF of Mahimahi emulation and live 5G networks. Mahimahi underestimates the RTT and fails to capture the RTT distribution.

Emulating this variability is critical for testing latency-sensitive applications such as VR/AR and remote driving.

Mahimahi also emulates delay variability based on packet delivery traces. Despite using a fixed propagation delay, it must hold the packet until it sees a PDO before releasing it (Fig. 15). However, our measurements suggest that it fails to fully capture the base RTT variability. To quantify this error, we compared the packet RTT reported in live experiment with that of Mahimahi.

Specifically, we conducted repeated packet RTT tests and Mahimahi recordings individually over live networks, following the randomized trial approach (§5.4.3). The packet RTT test involves a client sending a 1400-byte UDP packet (roughly an MTU-size) every 50 ms to our echo server and noting each packet’s RTT. We repeated this test 10 times, and both the RTT test and Mahimahi recording session lasting 60 seconds. Next, we ran the exact same packet RTT test under Mahimahi’s emulated interface, using the recorded trace and setting the propagation delay to half of the minimum RTT from the live packet RTT tests.

Figure 16 shows the cumulative distribution function (CDF) of packet RTTs on the live network and Mahimahi replay. It indicates that Mahimahi underestimates packet RTT (by 16.88% and 13.25% at the median for T-Mobile and Verizon, respectively), and

its distribution differs from that of the live network (as seen in the shape of the CDF curve). This suggests that simply increasing Mahimahi’s fixed propagation delay (i.e., shifting the CDF curve to the right) does not capture the variability. Note that this experiment was performed under stationary conditions with a strong signal, where network performance is more stable. In a mobile scenario, where packet RTT can vary more or even change, Mahimahi’s fixed propagation delay approach may perform even worse.

This is because PDOs, in principle, only *partially* capture base delay changes. Figure 17 illustrates a case where the packet base delay changes at t_2 from 20 ms to 40 ms, and Mahimahi fails to apply the correct delay for a certain packet. Note that base delay changes may occasionally occur in live cellular networks due to factors such as increased retransmission delays caused by a weakened radio signal. In this illustration, during the recording phase, the four packets delivered from t_1 to t_2 experience a 20 ms base delay, while packets from t_2 to t_3 experience a 40 ms base delay. The receiver indeed perceives a delay change since, after receiving four packets, it does not receive any packets for 20 ms before receiving the next set. This 20 ms “blackout” period is also reflected in the PDO trace during the replay.

A PDO blackout means no packet delivery. Any packets scheduled for delivery during this period will be delayed until the next available opportunity. As a result, only packets arriving during the blackout period (relative to the link emulator) will experience a delay, while others will *not*. However, sparse workloads, such as those in Fig. 17b, may have packets arriving outside the blackout period and thus not experiencing any delay.

Conclusion. Fixed propagation delay and PDOs are insufficient to model cellular network delay variability. Therefore, we need to record packet RTTs over time through probing during the recording phase and apply time-varying delays during the replay.

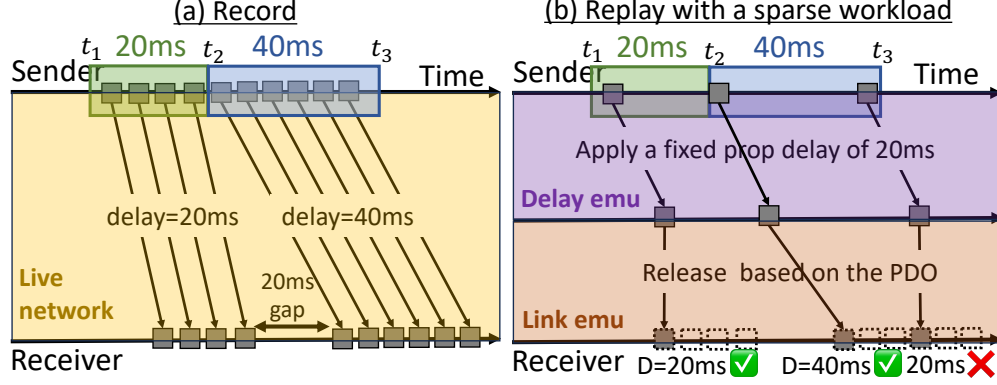


Figure 17: Mahimahi PDO approach fails to capture the base delay changes for sparse workload.

5.2.2 Performance depends on workload

Recall that Mahimahi uses a Saturator to ensure that the network always has packets ready to send, and so any available PDOs will be consumed and recorded. Then, a subset of those PDOs is used when replaying any given workload. An underlying assumption is that the same PDOs would have been available for the replayed workload. However, our experiments on live cellular networks show that *the network substantially changes the PDOs it provides depending on the workload*. We reached this conclusion upon observing that short flows consistently experience lower bandwidth than longer flows.

To demonstrate this, we conducted live experiments in which our server periodically sent *packet trains* to a client. Each train consists of N back-to-back UDP packets, each 1400 bytes, followed by a 100 ms gap—long enough to clear out any packets from previous trains). We refer to N as the train size. Each packet in a train is tagged with a train number and a sequence number reflecting its order within a train ($P_0, \dots, P_{N-1}$). The client then records each packet’s arrival time. Additionally, the client sends a 100-byte ACK back to the server upon receiving the last packet of a train (P_{N-1}). The *train completion time* (TCT) is defined as the time at which the server receives the ACK minus the time it sent P_0 . We performed this test with different train sizes following our randomized trial approach (§5.4.3). Each test lasted 5 seconds, and there are 12 tests

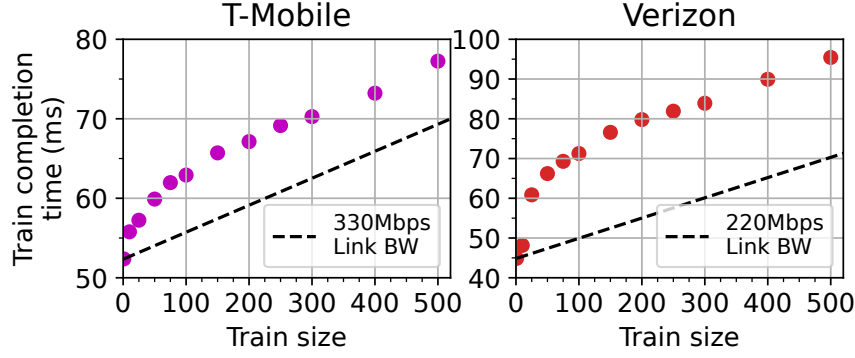


Figure 18: Mean train completion time (TCT) of different sized trains (N), and TCT if trains were delivered according to the Saturator’s observed bandwidth.

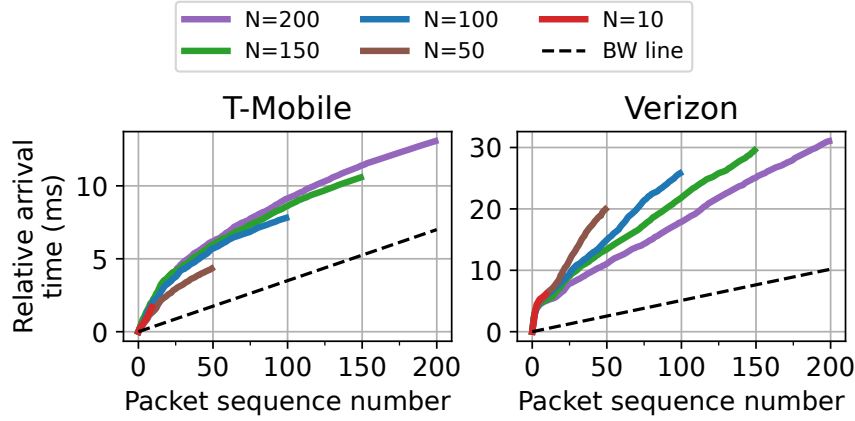


Figure 19: Mean relative arrival time for each packet of different sized (N) trains.

with different train sizes (1, 10, 25, 50, ..., 500) in one trial. We repeated this trial 50 times. Finally, we also recorded network performance with the Saturator, as Mahimahi would.

Figure 18 shows the mean TCT for each train size on T-Mobile and Verizon 5G. The dashed black line represents the TCT if the link had a fixed bandwidth equal to that observed by the Saturator (equivalent to the mean TCT with Mahimahi replaying the Saturator). If network performance were independent of workload, then the mean bandwidth would remain the same for all train sizes, and the mean TCT would follow a linear function of the amount of data being delivered, i.e., a linear function of N . In particular, it should coincide with the bandwidth observed by Saturator. However, the

observed TCTs do not conform to a straight line and generally do not match the Saturator line, with TCTs being up to 11.5% higher than the Saturator in T-Mobile and 35.8% higher in Verizon. This indicates that *the service experienced by the train workloads is consistently and significantly different than the service experienced by the Saturator’s heavy workload.*

To better understand these observations, we examined the arrival times of packets within each train, revealing the PDO patterns recorded by different trains. For each packet $P_i \in \{P_0, \dots, P_{N-1}\}$ in each train, we calculated its *relative arrival time* as $t(P_i) - t(P_0)$, where $t(\cdot)$ denotes the receiver’s observed arrival time of a packet. We present the mean relative arrival time as a function of packet sequence number i for different train sizes N in Figure 19.

These results confirm that the link’s rate (or PDO) depends on the workload. The network provides a lower delivery rate for the first few packets in any train (note that in these plots, a higher slope indicates a lower delivery rate). As the train progresses, the delivery rate increases, and the slope begins to approach that of the Saturator. We also repeated the same packet train tests in the opposite (uplink) direction but found that the link’s rate remains uniform regardless of N (i.e., it follows a straight line). We suspect that the rate-workload dependence in the downlink results from the operator’s proprietary packet scheduling implementation. For example, cellular network packet scheduling may depend on historical application traffic and the current queue depth when scheduling packets [110]. This rate-workload dependence was observed across all conditions tested in §5.4, including both peak and off-peak hours.

Interestingly, T-Mobile and Verizon have dramatically different implementations. T-Mobile’s delivery rate approaches the Saturator’s rate (slope) as i increases, mostly regardless of N . Verizon’s delivery rate, on the other hand, asymptotes to significantly different values depending on N , with larger N approaching the Saturator’s heavy-workload delivery rate more closely. Additionally, in T-Mobile, the first 50 packets

of trains with $N > 50$ are delivered more slowly than the 50 packets of the train with $N = 50$, whereas Verizon shows an inverted behavior. This further complicates record-and-replay, as we aim for a general and relatively accurate approach across different operators and locations.

We also found a more minor way in which performance depends on workload: the RTT of a packet varies with packet length by an amount not explained by throughput. For instance, based on our measurements on Verizon, the RTT of a 100-byte packet is 6.8 ms faster than that of a 1400-byte packet, even though the difference in serialization time at the bottleneck link rate (60 Mbps) should have been only ≈ 0.17 ms. This outcome aligns with findings from the prior latency study on 5G [91] and may be attributed to the additional time required for reassembling larger data chunks. Due to space constraints, we omit detailed results.

Conclusion. Cellular network performance can depend significantly on the workload. Our observations indicate that cellular providers allocate delivery rates (i.e., bandwidth or PDOs) differently for *light* and *heavy* workloads. The Saturator forces the link into its heaviest workload mode, which generally increases the available bandwidth. Consequently, using Saturator’s PDOs for a lighter workload can result in consistent bias (flows complete faster than they should). This suggests that diverse workloads are needed to capture different PDOs, but choosing the right representative workloads remains challenging.

5.3 CellReplay design and implementation

5.3.1 Design overview

At a high level, we want to solve the problems of capturing time-varying base RTT (§5.2.1) and workload-dependent performance (§5.2.2) in both record and replay.

We begin with the latter problem (§5.2.2). To achieve highly accurate emulation, an

obvious solution is to record performance under different workloads. However, recording every possible workload is impractical and degenerates into simply testing the apps directly on the live network. We also aim for the recorded workload to be independent of the tested apps (§5.1.1). Therefore, we can only record a *limited* number of different workloads.

From §5.2, we observe that short and continuous traffic are handled differently, while medium-length flows exhibit performance somewhere between the extremes. Inspired by this observation, our key approach is to record two workloads chosen at the extreme points on the spectrum of traffic patterns: (1) *Packet train probing* to capture link PDOs under short and bursty load (light PDOs), and (2) *Saturator* to capture PDOs under heavy continuous load (heavy PDOs). These workloads capture the essential behavior of link rate differentiation under light and heavy flows. We used two phones to record both traces simultaneously and show in our evaluation that interference between them is limited in practice.

During replay, we leverage both PDOs to match the provided workload. When the application under test begins sending packets, we initially release the first sequence of packets according to the light PDOs and then transition to heavy PDOs as the packet sequence lengthens. After a certain gap in the workload, we return to the light PDO trace.

Returning to the problem of time-varying RTT (§5.2.1), we design the packet trains to avoid inflating queues, so that it gives us a good measurement of base RTT. The packet train probing serves a dual purpose: to record changing base RTTs (for any workload) and PDOs for shorter packet sequences.

Finally, the effectiveness of the above design depends on parameter choices. For example, a too-small train will not capture the network’s light workload behavior completely, forcing us to go to the heavy PDO trace too soon; if the trains are too large, we cannot sample frequently as network performance may then resemble that of a heavy

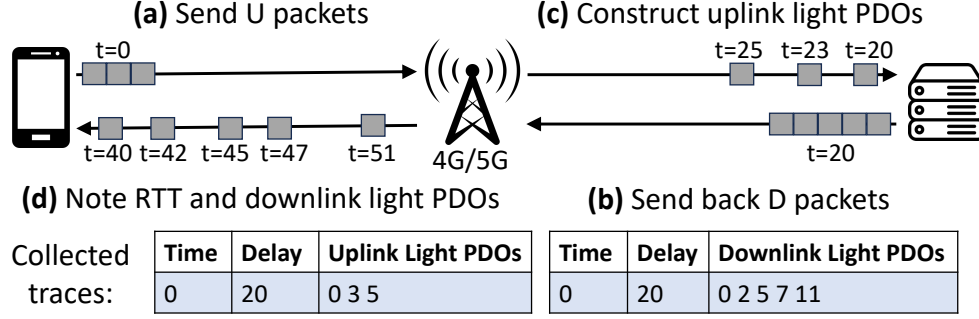


Figure 20: Packet train workload alongside with its recorded base delay and light PDO traces for uplink and downlink.

workload, and there is a risk of inflating base RTT measurements due to congestion. Thus, before recording, we conduct a calibration phase to determine train size, train gaps, and other parameters that will yield the least error.

In summary, CellReplay has *three* components. When recording network traces in a specific environment, we first perform an automated **calibration** of parameters in that environment, and then start **recording** live traces by running packet train probing and Saturator in parallel. These traces are then used to emulate the network during **replay**. The following subsections detail each of these components: record (§5.3.2) and replay (§5.3.3), before returning to calibration (§5.3.4), which is best understood after seeing the rest of the design.

5.3.2 Recording network traces

There are three time-series metrics we want to record: (1) base delay, (2) light PDOs, and (3) heavy PDOs.

Base delay and light PDOs. The base delay trace should reflect the network’s round-trip time (RTT) without any queueing delays introduced by the workload itself. Ideally, this trace would be captured by periodically measuring the RTTs of small packets. The one-way base delay can be estimated by halving the RTT.

Light PDOs can be captured by periodically sending a limited number of

back-to-back packets, i.e., a *packet train*, in both the uplink and the downlink. The number of packets should be small enough to capture the network’s light workload behavior and ideally some of the transition to moderate workload, without pushing the network into heavy workload mode. In particular, the train should be short enough to avoid “warming up” the network for the following train. As a result, both the base delay trace and light PDOs share similar requirements. We can collect both simultaneously using a packet train probing workload on a single device. This workload uses MTU-sized packets, as a significant amount of traffic is still required to capture the transition point between light and heavy modes.

Figure 20 provides an example of how this process works. In every G ms, (a) the client sends U back-to-back MTU-sized packets to the server. Upon receiving the first packet of the train, (b) the server sends back D back-to-back MTU-sized packets. The server also (c) records each packet’s arrival within that train and uses it to calculate the uplink light PDOs as the arrival time of each packet minus the arrival time of the first packet (since, during replay, the base delay will be added). When the client receives the corresponding downlink train, (d) it infers the current base RTT as the receipt time of the first downlink packet minus the send time of the first uplink packet (within that train). It then calculates the downlink light PDOs based on packet arrival times, just as the server did.

Heavy PDOs. The heavy PDOs are collected using a Saturator (similar to Mahimahi) that saturates the link with packets beyond its bottleneck rate, effectively “requesting” the link to remain in max bandwidth mode. In practice, we developed our own Saturator tool, which sends MTU-sized packets in both the uplink and downlink at fixed upload and download rates, eliminating the need for two phones as in Mahimahi’s Saturator [18]. We overestimated (by 25%) the max link bandwidth measured using an existing bandwidth test application like iperf or speedtest. We confirmed that the reported

throughput from our Saturator is similar to UDP iperf.

However, running both Saturator and packet train probing on a single device is not feasible, as the Saturator will overload the queue, leading to two issues: inflating the base delay measurement and keeping the link in maximum bandwidth state. One solution is to run these workloads in separate trials, which may be permissible under stationary conditions but is less ideal under mobility. Alternatively, we chose to perform these workloads on separate identical phones placed in close proximity. This is possible since most (if not all) cellular network providers employ user-separated queues [18] such that the Saturator traffic will not inflate the packet train probing measurement results. Beyond the known separation of queues, we confirmed that light vs. heavy bandwidth allocation is also separated on both Verizon and T-Mobile: when one phone runs the Saturator, the other phone running packet train probing still experiences light-workload service.

Note that the two-phone method is not without limitations. The phones may not always connect to the same base station all the time, especially in a mobile environment where handoffs could occur slightly differently. We leave this discrepancy for future work.

5.3.3 Replaying network traces

CellReplay takes input traces of base delay, light PDOs, and heavy PDOs over time to emulate network performance in a virtual interface. At a high level, CellReplay first applies a base delay to each packet based on the delay trace, adjusts the delay for any latency offset from packet-size calibration (§5.3.4), and then releases packets according to either the light or heavy PDOs.

In more detail, CellReplay operates in two states: active and inactive. Initially, CellReplay is in the inactive state until it receives a packet at some time t , relative to the start of the emulation. This event triggers CellReplay to enter the active state, which involves preparation as shown in Figure 21. CellReplay searches for the most recent base

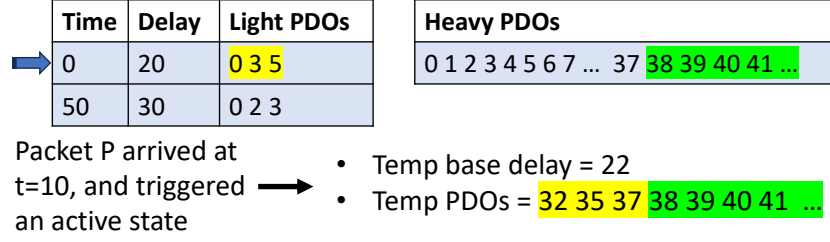


Figure 21: Temporary base delay and PDOs constructed when a CellReplay first enters an active state.

delay ($DELAY$) and light PDOs ($LightPDO$) where the timestamp is $\leq t$. Since the trace is sampled per G , linear interpolation is used to assign $DELAY$ to packets arriving between two samples. CellReplay then saves $DELAY$ and constructs temporary PDOs ($TempPDO$) by adding every PDO entry in $LightPDO$ with $t + DELAY$. It then concatenates these with the suffix of the heavy PDOs, starting from $t + DELAY + \max(LightPDO) + 1$. The system is now done entering the active state.

As long as the system remains in the active state, packets are initially delayed by $DELAY$ plus a size-based delay compensation $comp(size(P))$. As discussed in §5.2.2, base delay may depend on packet size; the specific adjustment $comp(\cdot)$ is determined during calibration. $DELAY$ and $TempPDO$ remain unchanged unless the system enters an inactive state¹⁰. After a packet is delayed, it is either placed in a PDO queue or dropped if the queue exceeds B bytes. Packets are dequeued according to the time schedule in $TempPDO$ using byte-wise dequeuing. This process mirrors Mahimahi’s PDO replay, with CellReplay using the temporary (concatenated) PDO trace. As a result, early packets in the active state will experience light PDOs, while later packets will experience heavy PDOs. Once F milliseconds pass without any packets in the PDO queue, CellReplay returns to the inactive state. Any future arriving packet will then

¹⁰Replaying base RTT changes during the active period could lead to double-counting, as PDO traces already capture some RTT changes in the form of gaps between PDOs. While this might seem counterintuitive to our goal of emulating time-varying base delay, the issue discussed in §5.2.1 only arises when there is a gap in the workload packets that empties the queue. However, in such cases, CellReplay has an opportunity to re-enter the inactive state and select another base RTT. Although there is a slight chance that the workload gap is shorter than the inactive-state timer F , F is typically small (e.g., 5 milliseconds), so we did not find this to be a major problem in practice.

Table 10: Parameters set in calibration phase.

Parameter	Definition
U	Number of packets per uplink train
D	Number of packets per downlink train
$G_{\min}$	Lower bound of gap between trains (milliseconds)
F	Fallback timer to return to inactive state
$\text{comp}(s)$	Delay compensation for s -byte packets
B	Bottleneck buffer size in bytes

trigger the procedure to reenter the active state, as described above.

5.3.4 Parameter Calibration

We describe how to select values for the parameters in Table 10. The parameters U , D , $G_{\min}$, F , and $\text{comp}(s)$ are exclusive to CellReplay and are calibrated in every new environment before recording traces. This process is automated. B is a standard network emulation parameter, derived using a classical max-min approach [118].

Setting U and D . We profile the network to determine a *packet train size* that provides the best overall approximation of the network across other sizes. We first conduct randomized experiments with different packet train sizes (the same as §5.2.2) using a fixed train gap that is conservatively large enough to ensure the link returns to its light-workload state. In our implementation, we use a 100 ms gap, and the set of train sizes we consider is $\{5, 25, 50, 75, \dots, X\}$ where X is chosen such that the resulting mean sending rate (including gaps between trains) is half of the bottleneck throughput.

After running for 10 trials, we compute $R_N(i)$ for each train size, which represents the mean relative arrival time of the i -th packet in an N -packet train. The relative arrival time is the packet’s arrival time minus that of the first packet in its train. R_N essentially represents the mean light PDOs of an N -sized train. We further define $R_N^*(i)$ as the estimated mean arrival time of the i -th packet *in replay mode*, assuming we choose to record trains of size N . More specifically, recall that during replay, we follow light PDOs

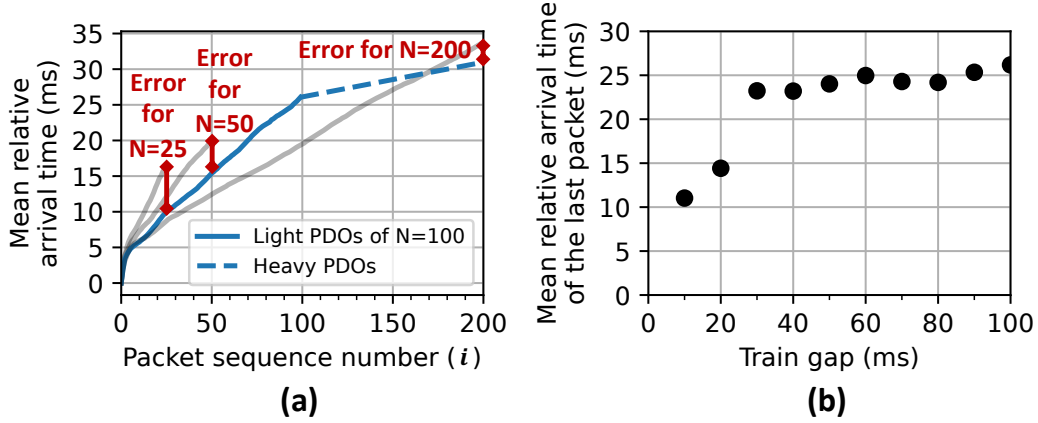


Figure 22: (a) Estimating error if a 100-sized train is used to estimate $\{25, 50, 200\}$ -sized trains. (b) G_{min} is chosen as the smallest train gap that results in performance similar to a large train gap (i.e., 30 ms).

before splicing in the heavy PDOs; therefore, $R_N^*(i) = R_N(i)$ for $i \leq N$, and otherwise, $R_N^*(i) = R_N(N) + heavy(i - N)$, where $heavy(x)$ is the delivery delay of x packets based on the mean throughput of the heavy workload.

The purpose of $R_N^*(i)$ is to help us calculate the estimation error of $R_N(N)$ (i.e., the mean relative arrival time of the *last* packet of an N -packet train) for every other train size N that we have tested. Let L be the train size used to estimate the error for other trains. Fig. 22(a) shows the estimation error when using $L = 100$. The blue line is $R_{100}^*(i)$, which represents the PDOs based on the concatenation of mean light and heavy PDOs. If we use that to estimate $R_N(N)$ of *other* trains $N \in \{25, 50, 200\}$, the prediction will result in some error (shown in red). For each train size L , we compute the mean error over *all* tested train sizes, and our chosen train size is the L that yields the smallest mean error. We conduct the entire procedure in two directions (uplink and downlink) separately to choose the train lengths U and D .

Finding G_{min} . If the train gap G is too small, the link will not have enough time to reset to its light-workload mode before the next train arrives. We typically set $G = 50$ ms for stationary conditions and $G = 100$ ms for mobile conditions. However, in certain

environments, this may not be large enough. We thus aim to find G_{min} , the smallest value at which the link has enough time to return to its light-workload mode, ensuring that our chosen G is at least that value.

We conduct another randomized experiment, this time testing different train gaps. Again, we send sequences of trains; however, in this case, we fix the train length at our chosen value (U or D , for uplink or downlink, respectively, in separate experiments) and vary the gap g . We begin with a conservatively large gap (as in the previous experiment) and test gaps of decreasing size; in our implementation, $g \in 100, 90, 80, \dots, 10$ ms. Let $r_{last}(g)$ denote the mean relative arrival time of the last packet in trains with gap g . Intuitively, as g decreases to a too-small size, the link will begin staying in its heavy-workload mode, causing $r_{last}(g)$ to decrease. We set G_{min} as the smallest g for which $r_{last}(g)$ is within 20% of its value with the conservatively large gap, i.e., $r_{last}(100ms)$. In the example of Fig. 22(b), CellReplay selects $G_{min} = 30$ ms.

Inferring F . Recall that F determines how long CellReplay’s emulated link remains idle in the active state before transitioning back to the inactive state. We derive F using the same data collected to select G_{min} , which involves calculating the difference between G_{min} and the time required for the queue to clear, which is observable as $r_{last}(G_{min})$.

Assume that internally, the bottleneck link remains in its heavy-workload state while it has queued packets to send, and then once it remains idle for some time F , it returns to the light-workload state. With the gap of G_{min} , this state change is just barely reached. Thus, we have $S + G_{min} \approx Q + F$, where S is the time the sender takes to send the train, G_{min} is the time it waits before beginning the next train, and Q is the time for the link to entirely clear the packet train out of its queue. Since S and G_{min} are both timed in user space, S is close to zero. Furthermore, because we have assumed the link of interest is the bottleneck, Q is observable at the receiver as the time between receiving the first and last packets of the train, i.e., $r_{last}(G_{min})$. Thus, we can estimate

$F = G_{min} - r_{last}(G_{min})$. In Figure 22b, the G_{min} is 30 ms, and its $r_{last}(g_{min})$ is 23.23 ms. Thus, we compute $F = 6.77$ ms (and round it to 7 ms when setting the parameter).

Inferring $comp()$. We examine the impact of packet size on RTT by sending packets with sizes sampled uniformly at random from $\{100, 200, \dots, 1400\}$ bytes every 50 ms to a receiver that replies to each with a 100-byte ACK. After running for a configurable time (5 minutes for our case), we calculated the mean RTT for each packet size x , denoted as μ_x . The compensation delay for a packet size s , or $comp(s)$, is computed as follows. If $s < 100$, $comp(s) = \mu_{100} - \mu_{1400}$. Otherwise, $comp(s) = (1 - \alpha)\mu_i + \alpha\mu_j - \mu_{1400}$ where i and j are the tested packet sizes immediately less than and greater than s , respectively. The parameter α is set to linearly interpolate between the two observed means, i.e., $\alpha = (s - i)/(j - i)$. The compensation is always relative to 1400-byte packets because that is the size used to measure base RTT. This procedure is repeated separately for uplink and downlink measurements, with sender and receiver roles swapped.

Inferring B . We use a classical max-min approach [118] to infer the bottleneck buffer size using the difference between the minimum and maximum RTT of packets under heavy load. Specifically, we run iperf and monitor it with tcpdump. The buffer size (B) is calculated as $(RTT_{max} - RTT_{min}) \cdot C$, where C is the observed link capacity [119].

5.3.5 CellReplay implementation

CellReplay's record client is implemented in Java to ease porting into Android. Meanwhile, the record server is implemented in Python 3. The client accepts workload configurations as user inputs, which are determined from the automated calibration performed separately. These configurations are then sent to the CellReplay server. For packet train workloads, the user must input: the number of packets per uplink and downlink train, and the gap between trains. Saturator workload requires the maximum upload and download bandwidth, which can be determined using other bandwidth

probing tools (e.g., `speedtest.com` or `iperf`). Both client and server send 1400-byte UDP packets via a socket.

CellReplay’s replay is built on top of the Mahimahi shell, extending all of Mahimahi’s core functionality. This includes the ability to run unmodified applications, or the option to nest the shell with another Mahimahi shell, such as the HTTP ReplayShell. The CellReplay shell accepts the base delay, light PDO trace, and heavy PDO trace as inputs. Similar to the Mahimahi network emulator shell, CellReplay controls a virtual network device (TUN) and captures all IP datagrams from an unmodified application running inside the shell. It then delays each packet before sending it to another interface, such as loopback or Ethernet.

5.4 Evaluation

Our goal is to evaluate the accuracy of CellReplay’s emulation in replicating application performance compared to its live network counterpart. We also compare CellReplay with Mahimahi [72]. The evaluation includes experiments that test CellReplay’s accuracy across (1) different networked applications, including web browsing and random file transfers using TCP, (2) different cellular providers and technologies, including T-Mobile, Verizon, 5G mid-band, and 5G low-band, and (3) different environmental conditions, such as good signal strength, weak signal strength, crowded areas, and various mobility levels (stationary, walking, and driving). The details on the environments and their calibration parameters are listed in Table 11. Finally, we present a use case of using CellReplay and Mahimahi to evaluate ABR algorithms.

5.4.1 Experimental setup

We designed two test setups: a *live network* and an *emulation* test setup, as shown in Figure 23. The *live network* test setup was used for running application tests on the live network. During the tests, we tethered a laptop to phones connected to 5G or 4G

Table 11: All tested conditions that includes stationary, walking, and driving scenarios. We tested TM and VZ networks under 5G mid band (MB) and low band (LB). We also showed the used CellReplay configurations.

Name	Description	Op.	Net. type	CellReplay config
stationary-good	UEs were in an office near a window	TM	5G MB	U=25, D=75, G=50, F=5
		VZ	5G LB	U=10, D=100, G=50, F=7
stationary-crowded	UEs were in a crowded library during rush hours	VZ	5G LB	U=10, D=100, G=50, F=7
stationary-weak	UEs were in a basement of a building with no window	TM	5G LB	U=25, D=75, G=50, F=5
walking	User was walking through an office corridor in a loop while holding the UEs	VZ	5G LB	U=10, D=50, G=100, F=7
driving	UEs were with the user driving in a loop around the university area	TM	5G MB	U=10, D=75, G=100, F=5

networks. The client application (e.g., a web browser) and the application server (e.g., a web server) communicate via a UDP tunnel (based on [112]). We deployed our server (i.e., the remote endpoint) in the same geographical area—within 10 miles of the phones—to minimize the network path length and, consequently, reduce the likelihood of experiencing congestion over long paths. We also used a similar setup, albeit without a tunnel, to record the cellular network traces using UDP traffic. However, instead of a single phone, we used two identical phones to separately perform packet train probing and the Saturator workload.

We used the *emulation* test setup to test applications under an emulated network interface that employed either CellReplay’s or Mahimahi’s replay approach. Although this setup is similar to the live network test, we made two modifications: (1) we replaced the USB-tethered interface with a direct high-speed Ethernet connection to our server via a single switch, and (2) we ran the tunnel inside either CellReplay’s or Mahimahi’s replay network-emulation shell, which emulates the network using recorded traces. The same client and server devices were used for record and replay.

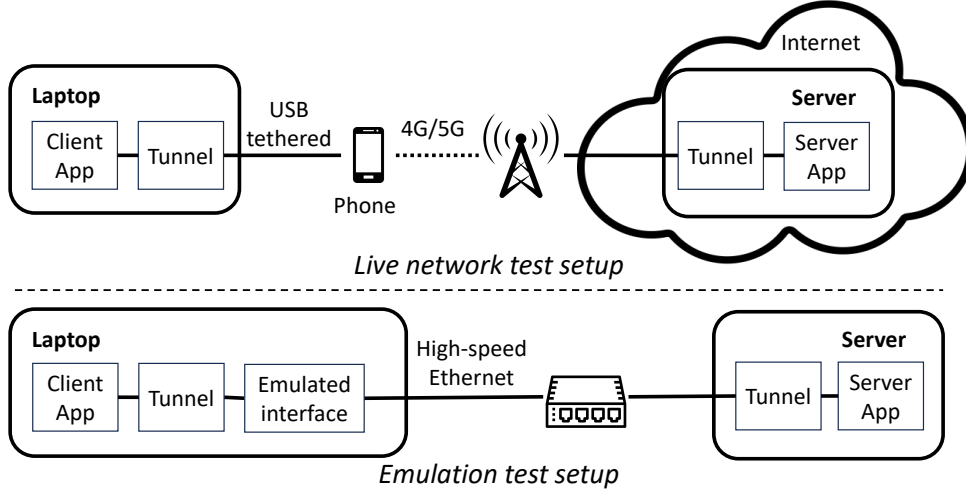


Figure 23: Live network and emulation application test setup.

End-point specifications. We used two Samsung Galaxy S22 (SGS) and two Google Pixel 5 (Pixel) phones for testing. We had an unlimited data plan from both T-Mobile and Verizon. Since we observed no performance difference between the SGS and Pixel across different operators, we connected our SGS devices to T-Mobile and our Pixel devices to Verizon for convenience. The laptop in our setups featured an Intel Core i7 CPU, 16 GB RAM, and a 512 GB SSD, running Ubuntu 20.04.

5.4.2 Applications under Test

We tested CellReplay’s fidelity using two real-world applications: (a) web-page loads, which exhibit complex traffic patterns and represent the most popular application type on mobile devices, and (b) random file downloads, which have simpler traffic patterns but involve large-sized flows.

Web page load test. Our client ran a Chromium browser to load a web page from our server. To generate the list of web pages for our testing, we began by randomly selecting 200 internal and landing pages from the Hispar list [73]. Using Chromium, we loaded all pages and recorded each page’s total size of the compressed web objects. We then sorted the pages based on this total size and selected five pages each for landing and

Table 12: List of web pages used for testing along with details on the page composition, including the number of objects (“#objs.”), and the mean (“avg. sz.”) and total (“tot. sz.”) compressed object sizes (both in KB). “PT” indicates page type.

PT	ID	URL	#objs.	avg. sz.	tot. sz.
<i>Landing</i>	L1	bing.com	2	205.53	411.06
	L2	microsoft.com	36	24.66	887.64
	L3	dropbox.com	76	22.15	1683.49
	L4	glassdoor.com	64	43.23	2766.46
	L5	discord.com	37	172.86	6395.86
<i>Internal</i>	I1	en.wikipedia.org/wiki/Naivety	19	13.51	256.64
	I2	box.com/about-us	48	14.39	690.95
	I3	etsy.com/payments	50	28.92	1446.22
	I4	youtube.com/user/ESPN	44	56.88	2502.53
	I5	colubrina.tumblr.com	69	116.57	8043.10

internal pages corresponding to the 10th, 30th, 50th, 70th, and 90th percentiles of the distribution. We use ‘L-ID’ and ‘I-ID’ to refer to an individual landing and internal page, respectively. ‘ID’ is a number indicating the order based on page size, where lower values represent smaller page sizes. The list of web pages used for testing and its details on page composition is detailed in Table 12

We utilized Mahimahi’s HTTP record-and-replay framework [72] to replay these pages, ensuring that the fetched content for a given page remained consistent across all trials. First, we recorded all HTTP requests and responses using `mitmproxy` [74], following Mahimahi’s record format, while loading a web page using a headless Chromium browser. Next, we used Mahimahi’s ReplayShell to serve the responses over the HTTP/1.1 (using Apache2) or HTTP/2 (using h2o from the [75] extension).

In our test setup (Fig. 23), ReplayShell was deployed on our server, while the laptop ran a headless Chromium browser to repeatedly fetch a web page. We cleared the browser and DNS caches before every web-page fetch, and we used TCP Cubic, the default TCP implementation on Linux, for this test. Lastly, we measured *page-load time* (*PLT*), based on the `onLoad` event [76], as the application performance metric.

Random file download test. For this test, we implemented a client that sends download requests (each fewer than 10 bytes) to a server. Each request selects a file uniformly at random from a list of files of varying sizes, and the server delivers the corresponding file data. After completing the transfer, the client then sleeps for 50 ms before requesting another file. We implemented the client and server in Python and used TCP Sockets. We measured the *file download time* as the turnaround time between when the client sends its request and when it receives the requested file.

Adaptive video streaming over HTTP. We ran a client-server setup from [95], [120], which includes multiple adaptive bitrate (ABR) streaming implementations. However, instead of using the 2K video from that setup, our server hosts a 250-second-long AVC 4K video (*indoor soccer* from [121]) with a 4-second chunk duration encoded with 12 bitrates of [100, 200, 375, 550, 750, 1000, 1500, 3000, 5800, 7500, 12000, 17000]. We used a Chromium browser to run the video player.

5.4.3 Methodology

Randomized trials. When evaluating a networked system’s performance on the CellReplay emulated interface versus the live network, we must account for cellular network variability. Thus, all our experiments used *randomized trials*. Each experiment comprised multiple trials, with each trial consisting of multiple tests and recording sessions. A test involved running an application (e.g., loading a web page) on a live network, while a recording session (e.g., CellReplay record) gathered a trace of the live network. The tests and recording sessions ran for the same duration, and we randomized the sequence of tests and sessions within each trial. We then compared the application’s performance (across multiple tests) on the live network to that on the emulated network to calculate the emulation accuracy.

Calculating the emulation accuracy. We quantified emulation error using the normalized difference between two distributions of application performance: one from the live network and the other from the emulation. To measure the difference between these distributions, we used Earth Mover’s Distance (EMD) [122], defined as:

$$EMD(L, T) = \int_{-\infty}^{+\infty} |L(x) - T(x)| dx,$$

where L and T are the cumulative distribution functions (CDFs) of the observed application performance on the live and emulated networks, respectively. A lower EMD value indicates a high-fidelity emulation, meaning the performance distributions are more similar. Finally, we calculated the *emulation distribution error* by dividing the EMD with the mean (performance) value from the live network tests.

Other details. In each new network environment (or network operator), we first performed a calibration. To record CellReplay traces, we used one phone (*primary*) for packet train probing and the other (*secondary*) for running the Saturator. The *primary* phone was also used for calibration, testing applications on the live network, and recording Mahimahi traces using our own Saturator. For propagation delay, we provided Mahimahi with the minimum base delay observed from the packet train workload. During data collection, both phones (UEs) were held together at a close distance (≈ 12 inches). We also confirmed that both probing devices were attached to the same cell in all tests, except in the driving case.

5.4.4 Microbenchmarks

We first evaluated how accurately CellReplay emulates the time-varying base RTT and the non-uniform delivery rate in a cellular network. We conducted packet RTT and packet train tests (refer to §5.2.1 and §5.2.2) as separate experiments. Each experiment consisted of 30 randomized trials, with each trial including two record sessions (CellReplay and Mahimahi) alongside packet RTT and packet train tests. The sessions

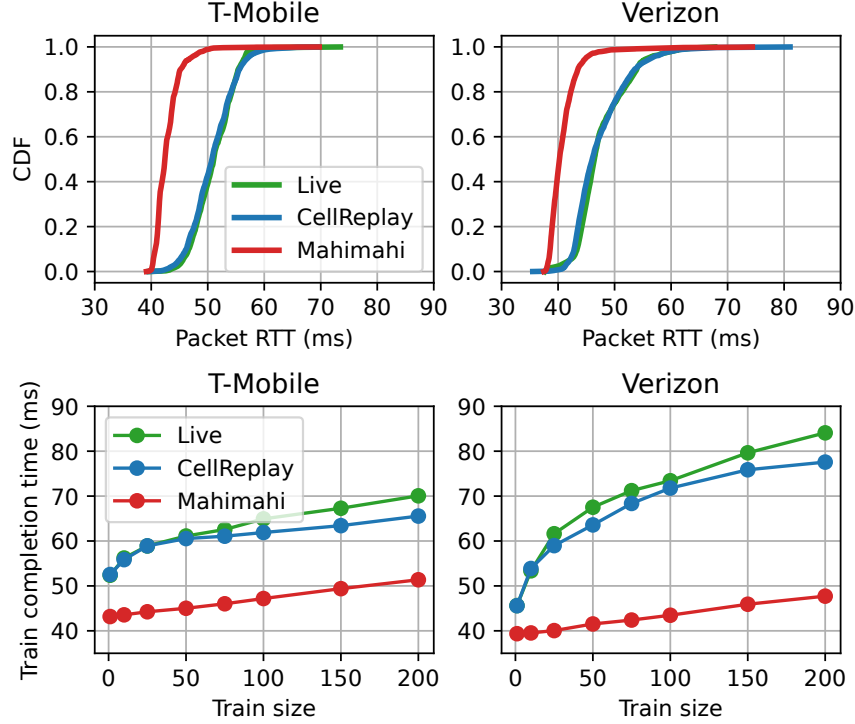


Figure 24: The CDF of RTTs (above) and mean TCTs (below) from the packet RTT and packet train tests, respectively, run on both live network and emulations.

(as well as tests) lasted for 10s. All tests were performed under “good” network conditions, i.e., UEs were held stationary near a window.

First, CellReplay accurately records RTT changes over time, with the CDF of packet RTTs closely overlapping with the live network (Figure 24). As expected, Mahimahi persistently underestimates RTT (e.g., median RTT is underestimated by 16.88% on T-Mobile and 13.25% on Verizon) and produces an RTT distribution that deviates from the live network. Second, CellReplay captures the non-linearity in train completion times as train size increases more accurately than Mahimahi, thanks to its light and heavy PDOs approach. The packet train experiment demonstrates CellReplay’s interpolation error when using a pre-defined train size (e.g., 75 packets for T-Mobile) for workloads ranging from 1 to 200 packets. For trains longer than 75 packets on T-Mobile and 100 packets on Verizon, CellReplay achieves lower interpolation errors than Mahimahi’s single (heavy) PDO approach, reducing errors from 26.68% to 6.44% on T-Mobile and

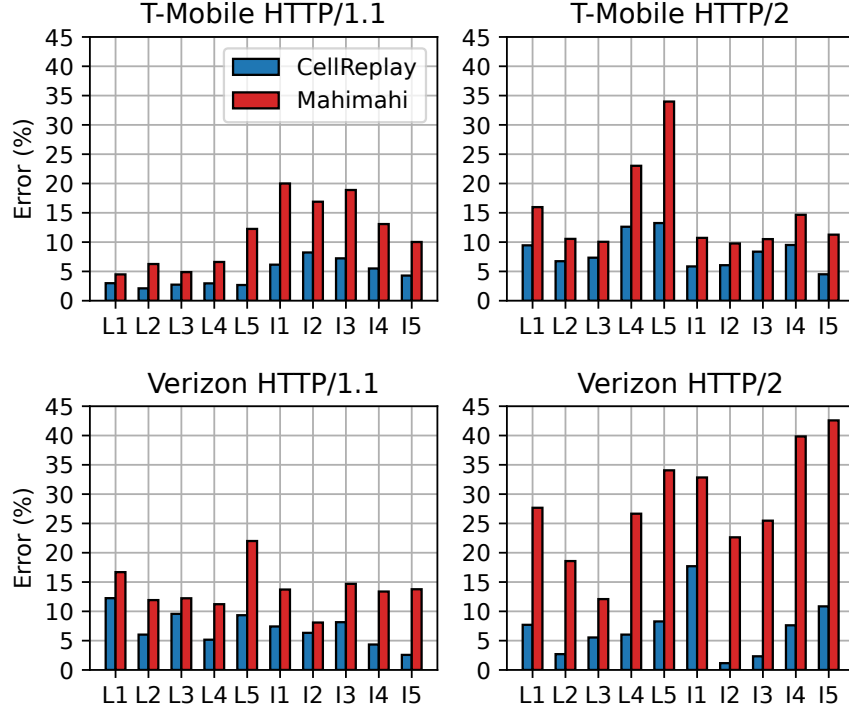


Figure 25: Emulation distribution error across different web-page load tests with HTTP/1.1 and HTTP/2.

from 43.24% to 7.74% on Verizon for a train size of 200. CellReplay’s interpolation errors for longer trains highlight an opportunity for future improvement.

When running these experiments, the primary phone recorded the base delay while the secondary continuously sent packets; the results, hence, suggest that the two-phone setup had minimal interference.

5.4.5 Web browsing test

We evaluated CellReplay’s accuracy for the web browsing by fetching both landing and internal pages (listed in Table 12) using HTTP/1.1 and HTTP/2. As before, we conducted these tests under “good” network conditions, with the UEs placed near a window. For each operator, we conducted four experiments, with each randomized experiment consisting of 10 trials. Each trial included fetching five pages using the live network and two recording sessions—one for CellReplay and the other for Mahimahi.

Each test and recording session lasted for 60 s. In total, we spent approximately 9.33 h conducting these experiments.

CellReplay outperforms Mahimahi across all page load tests (Figure 25). It achieves an emulation distribution error between 1.2%-17.7%, with a mean error of 6.7%. In contrast, Mahimahi has errors ranging from 4.5% to 42.6%, with a mean error of 17.1%. On average, CellReplay reduces emulation error by 60.8% for web-browsing apps compared to Mahimahi. Page load traffic is typically dominated by small object transfers, which are sensitive to RTT and categorized as *light* traffic. Mahimahi’s underestimation of RTT and overestimation of PDOs from the Saturator lead to significant errors for small flows (see §5.4.6). All Mahimahi’s errors stem from underestimating *all* PLTs, with a mean PLT of 2637 ms compared to 2918 ms in real networks. Meanwhile, CellReplay accurately captures the network RTT and provides a better representation of real available bandwidth for these transfers, although some error still persists.

Below, we examine two dimensions to highlight CellReplay’s edge over Mahimahi.

Across different operators. CellReplay maintains low mean errors for both operators: 6.4% on T-Mobile and 7.1% on Verizon. Mahimahi has a mean error of 13.2% on T-Mobile and an even higher 21% on Verizon. Mahimahi’s poor performance on Verizon is expected: Verizon tends to assign small bandwidth for small file transfers (see §5.4.6), and page loads primarily involve fetching small objects.

Across different protocols. CellReplay performs well on both HTTP/1.1 and HTTP/2, with mean errors of 5.8% and 7.7%, respectively. In contrast, Mahimahi shows poorer accuracy, with a mean error of 12.6% on HTTP/1.1 and an even higher 21.6% on HTTP/2. The multiplexing behavior of HTTP/2 may lead to a more complex traffic pattern compared to HTTP/1.1, leading to more frequent medium-sized flows. As a result, CellReplay often uses an interpolated PDO, increasing the error. For Mahimahi, the issue is even more pronounced, as it always applies the heavy rate, significantly

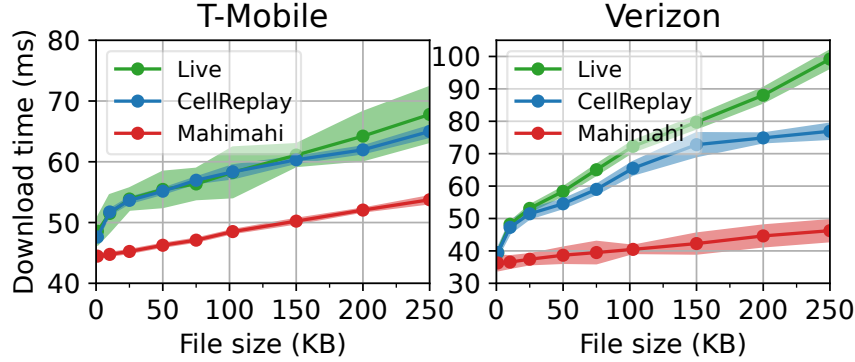


Figure 26: Mean small file download times along with its 95% CI.

underestimating flow completion times. Consequently, both methods experience higher errors for HTTP/2 than for HTTP/1.1.

5.4.6 Random file downloads test

To further evaluate CellReplay’s accuracy, we conducted randomized file download tests with small-sized files (1 KB to 250 KB) and medium-sized files (1 MB and 10 MB) on both T-Mobile and Verizon. Unlike the web-page loads, download tests are not affected by non-network-related computations (e.g., JavaScript parsing). Additionally, unlike the packet train test, which sends fixed train sizes over time, the download test is fully random: the client selects each file randomly from a predefined list. This live experiment consisted of 20 randomized trials, each including: two test workloads (small and medium file downloads) and two recording sessions (Mahimahi and CellReplay). Each test and recording session ran for 60 s, resulting in a total experiment duration of 80 min per network operator. All tests were conducted under “good” network conditions.

Small-sized files. Figure 26 shows the file download times for sizes between 1 KB and 250 KB. Consistent with the observation in §5.4.4, download times do not conform to a straight line. Mahimahi significantly underestimates (mean) download times, with errors ranging from 8.4% to 20.7% on T-Mobile and 7.9% to 49% on Verizon. CellReplay

Table 13: Mean file download time (in ms) of medium-sized files with its emulation distribution error.

		1MB	10MB
<i>T-Mobile</i>	Live	142.52	1056
	CellReplay	135.88 (7.82%)	978.18 (8.19%)
	Mahimahi	117.68 (20.17%)	847.45 (20.44%)
<i>Verizon</i>	Live	155.18	1069.25
	CellReplay	135.68 (10.46%)	974.98 (4.88%)
	Mahimahi	106.63 (26.5%)	762.3 (23.67%)

manages to capture and emulate the non-uniform bandwidth availability with its light and heavy PDOs, resulting in significantly lower mean download time errors: 0.5%-3.5% on T-Mobile and 0.2%-22.4% on Verizon. CellReplay’s error increases with larger file sizes (e.g., 250 KB on Verizon) as it must interpolate once packet sequence lengths exceed a certain threshold.

Medium-sized files. Table 13 shows the mean file download times for 1 MB and 10 MB files alongside its distribution emulation error for CellReplay and Mahimahi, compared to the live networks. CellReplay achieves a mean distribution error of 9.14% for 1 MB downloads and 6.54% for 10 MB downloads across both providers. In comparison, Mahimahi produces significantly higher errors of 23.35% and 17.06%, respectively, for the same file sizes. Surprisingly, Mahimahi’s error increases further for 10 MB files, despite expectations that large file transfers would be dominated by heavy PDOs (i.e., the Saturator rate). We suspect that Mahimahi’s inaccuracy stems from a combination of base RTT underestimation and available bandwidth overestimation.

5.4.7 Interpolation effectiveness

We evaluated the impact of CellReplay’s PDO interpolation on emulation accuracy by comparing it to two variants: CellReplay with only light PDOs (CellReplay-light) and CellReplay with only heavy PDOs (CellReplay-heavy) for web page loading and file

downloads. CellReplay-heavy resembles Mahimahi but incorporates variable base delay and delay compensation based on packet size. CellReplay-light does not transition to heavy PDOs when the light PDOs end; instead, it restarts from the beginning. We used T-Mobile and Verizon traces, as in §5.4.5 and §5.4.6, and reported the average error across both operators.

Figure 27 presents the results. For web browsing, CellReplay-light performs similarly to CellReplay with interpolation (i.e, our CellReplay) and beats CellReplay-heavy, as web browsing is a mostly light workload. For the 1 KB download, all three versions perform similarly since RTT dominates performance and bandwidth is less critical. As the workload increases to 10 KB, 100 KB, and 1 MB, bandwidth becomes a more significant component of download time, but CellReplay-heavy’s emulation of bandwidth is inaccurate in this regime, leading it to have higher errors. However, at 10 MB, CellReplay-heavy’s error decreases as the workload starts to resemble the Saturator. Meanwhile, CellReplay-light excels for smaller downloads (10 KB and 100 KB) but struggles with larger files (1 MB and 10 MB). This suggests that neither light nor heavy PDOs alone can accurately emulate the characteristics of the wireless channel.

Notably, the variable base delay and delay compensation in CellReplay-heavy reduce error to 11.67%, compared to Mahimahi’s 18.77%. The error is further reduced by incorporating PDO interpolation, as in our full CellReplay system, bringing the error down to 5.68%. These results confirm that interpolating between light and heavy PDOs significantly improves emulation accuracy.

5.4.8 Accuracy under challenging stationary network conditions

We evaluated CellReplay’s accuracy under two challenging network conditions. In this first condition, labeled as ‘stationary-weak’, UEs connected to T-Mobile were placed in a windowless basement inside a building. In the second condition, labeled as ‘stationary-crowded’, we placed the UEs (connected to Verizon) in a library during

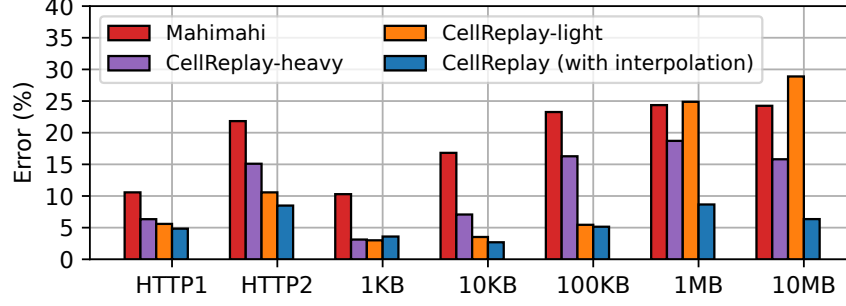


Figure 27: Emulation distribution error of CellReplay with and without light and heavy PDOs interpolation.

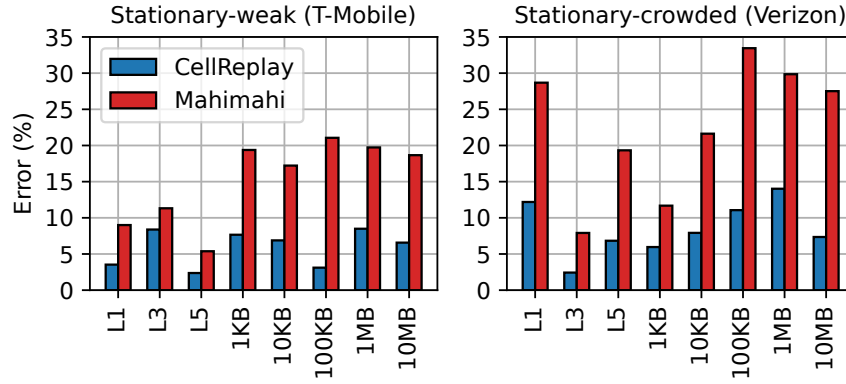


Figure 28: The web PLT and file download emulation distribution error when testing under challenging stationary cases.

crowded or rush hours. Both web-page loads and file downloads were tested under these conditions. To reduce the number of tests and minimize variance, we only selected pages corresponding to the 10th, 50th, and 90th percentiles of the HTTP/1.1 landing pages. We tested random file downloads with sizes ranging from 1 KB to 10 MB. The experiment was split into two parts: one for web-page loads and the other for file downloads. Each part consisted of 10 randomized trials. In the first part, each trial included loading three pages and two record sessions, while the second part had one random file download test and two record sessions per trial. Every test and record session lasted 60 s. The total experiment time under each network condition was 80 minutes.

Per Figure 28, across the two network conditions and providers, CellReplay has an average distribution error of 5.96% for web-page loads and 7.9% for file downloads.

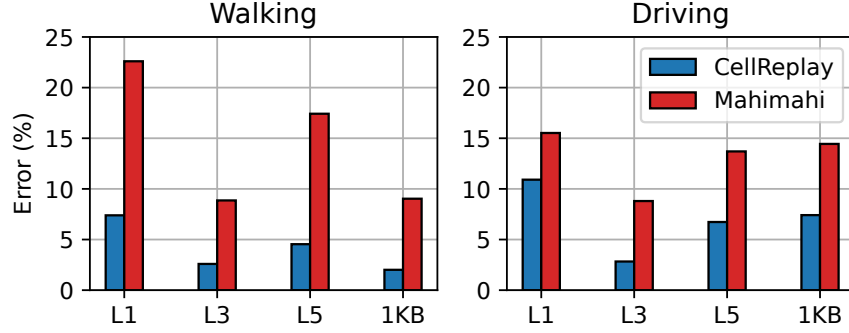


Figure 29: Emulation distribution error of three web PLTs and 1KB file download time under mobility.

Mahimahi, in contrast, has mean distribution errors of 13.63% and 22.02% for the same applications, respectively. Even under challenging conditions (**stationary-weak** and **stationary-crowded**), CellReplay still offers a respectable error rate and outperforms Mahimahi.

5.4.9 Accuracy under mobility

We also evaluated CellReplay’s accuracy under low and moderate mobility. For low mobility, a user walked through an office corridor in a loop while carrying the UE (connected to Verizon). For moderate mobility, the user drove around a university campus in a loop with the UE (connected to T-Mobile). We followed a fixed path while walking or driving in a loop to ensure consistent comparisons across trials. On average, a loop took 75 s while walking and 220 s while driving.

To reduce the number of tests, we performed web page load tests using only three HTTP/1.1 landing pages – L1, L3, and L5, which represent small, medium, and large web pages (§5.4.2), respectively. We also tested repeated 1 KB file downloads. To minimize variance, we limited each trial to two live tests and two recordings. We conducted two separate experiments: the first, with 1 KB file download and L3, and the second, with L1 and L5. We conducted 10 trials for **walking** experiments and 5 trials for **driving** experiments. Calibration was performed only once at the starting point. In

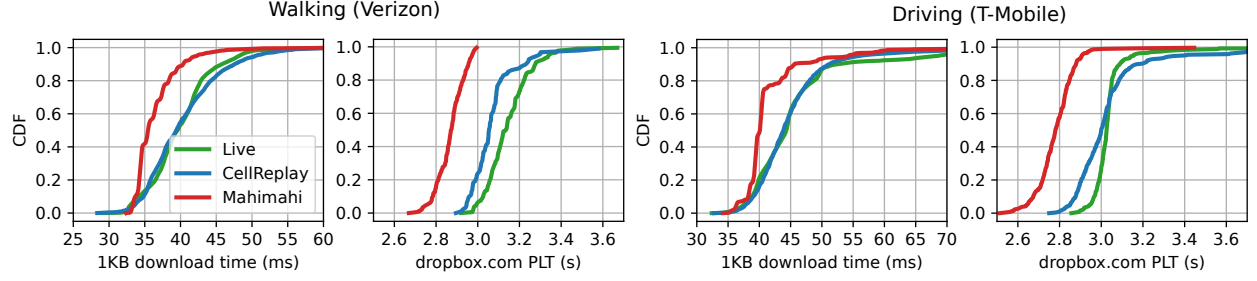


Figure 30: CellReplay manages to capture more application performance variability under mobility compared to Mahimahi. We cut the CDF graph as the tail is too long.

total, we spent 1.67 h walking and 2.45 h driving around the loop.

Figure 29 shows the emulation distribution error for CellReplay and Mahimahi. Network conditions are more variable under **driving**, which may also introduce packet drop, a factor that CellReplay does not capture¹¹. We indeed confirmed that **driving** triggered at least two handovers¹², and a few packets were dropped during certain periods of recording. Consequently, CellReplay’s accuracy suffers more under **driving** than **walking**. However, despite this limitations, CellReplay still provides a noticeable improvement (1.8 times) over Mahimahi in both mobility scenarios. This improvement stems from CellReplay’s ability to capture more performance variability than Mahimahi.

We plotted the CDF curve for 1KB file download times and L3 (dropbox.com) page load times in Figure 30 to get a closer look at CellReplay’s and Mahimahi’s performance under mobility. The CDF curve from CellReplay is closer to the live network’s curve compared to Mahimahi which suggests that CellReplay is able to replicate more performance variability than Mahimahi.

5.4.10 Use case: evaluating ABR algorithms

Finally, we demonstrate a common application for record-and-replay emulation: evaluating ABR algorithms for 4K video streaming. ABR algorithms are reactive,

¹¹This refers to the drop in IP packets. CellReplay can still capture the effects of the handover process through the increase in the base latency and PDO blackout and emulate it accordingly.

¹²There were a brief period (< 1 second) when the two devices were connected to different base stations due to the handovers were not done at the same time.

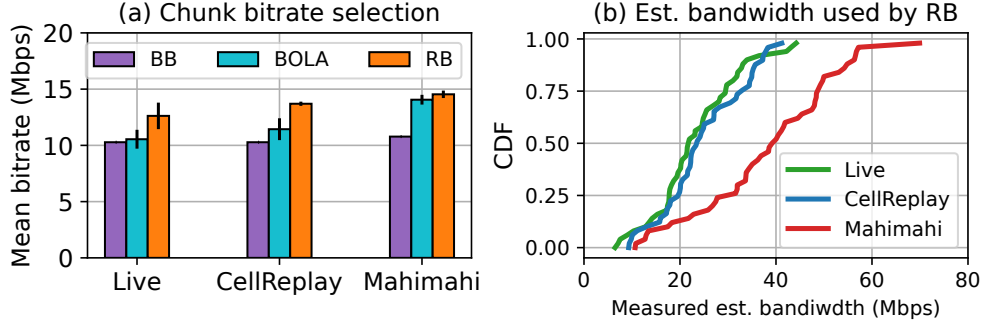


Figure 31: (a) Mean ABR bitrate selection result. Mahimahi shows positive bias to BOLA. (b) Reported estimated bandwidth from RB.

meaning their bitrate selection for video chunk downloads can be influenced directly (rate-based) or indirectly (buffer-based) by observed network performance. Thus, accurately emulating network conditions is crucial to avoid bias in algorithm evaluation.

We compared three ABR algorithms¹³ from [120] running on Verizon 5G under “good” network conditions using CellReplay and Mahimahi emulation: (1) Buffer-based (BB) and (2) BOLA, which make bitrate decisions based on buffer occupancy, and (3) Rate-based (RB), which estimates throughput from past chunk download times to select the next bitrate. Given the high download bandwidth reported by Saturator (270 Mbps), all three algorithms eventually select the highest bitrate (17 Mbps) and do not experience rebuffering. However, their startup phase [124] behavior differs, so we focused on bitrate selection for the first 10 chunks as our QoE metric. We conducted 10 randomized trials, each with three test workloads (streaming video using three ABRs) and two recording sessions (Mahimahi and CellReplay). Each test and recording session lasted for 30 s.

Figure 31a presents the mean bitrate results. As expected, Mahimahi reports a higher bitrate than Live-5G across all three ABRs, with an average overestimation of 17.73%, compared to 5.89% on CellReplay. This aligns with our findings that Mahimahi tends to overestimate network performance, leading to inflated application performance. But, more importantly, this also affects protocol evaluation. Mahimahi’s results suggest that

¹³This setup also includes MPC[123] and Pensieve[95], but these were not tested, as MPC requires modifying the hardcoded MPC table, and Pensieve requires retraining.

RB and BOLA are the best protocols, with BOLA significantly outperforming BB by 30.43%. However, in reality in this environment, BOLA performs significantly worse than RB and is similar to BB. CellReplay’s results are much more aligned with the live network. This discrepancy is largely due to Mahimahi’s overestimation of available bandwidth, particularly for smaller initial chunks, causing ABRs to quickly converge to the highest bitrate. In Figure 31b, we present the measured bandwidth estimation from RB, which shows that Mahimahi overestimates bandwidth by 44.38% in the median, compared to 7.96% on CellReplay. Unlike Mahimahi, CellReplay does not suffer from this bias and provides more accurate relative performance of ABRs due to its dual light and heavy rate approach.

5.5 Discussion on CellReplay and future work

We discuss CellReplay’s use cases, limitations, and plans for future improvements.

Use cases. As shown in §5.4.10, CellReplay can be used to evaluate new applications and protocols on cellular networks and provide more accurate emulation of real network performance compared to the state-of-the-art approach. CellReplay is superior for latency-sensitive applications, as it can emulate base delay variability, and applications with variable flow sizes, as it can emulate the bandwidth-workload dependency. Adaptive applications (e.g., ABR) that react to network measurement will also receive more accurate performance results with CellReplay.

We also provide traces that researchers and developers can use for testing on CellReplay. While recording traces with CellReplay requires a bit more effort than Mahimahi due to the use of two phones, this is a minor issue, as once a diverse set of traces is recorded, users can easily replay them without the phones (just as in Mahimahi). Since CellReplay’s implementation is based on Mahimahi’s shell, unmodified applications can easily use CellReplay emulated interface.

Inaccuracies in CellReplay. While we have made significant progress in faithfully replaying cellular performance, CellReplay involves several simplifications and assumptions: (1) CellReplay does not record and replay random packet losses (although it drops packets when the queue overflows and can be manually configured for a set random drop rate). We notice, however, that cellular links under stationary conditions are robust to random packet drops (e.g., due to packet corruption) due to link-layer retransmission [89]. However, packet drops are more frequent during handovers in mobility [125]. (2) CellReplay uses fixed calibration parameters before each recording session. A more adaptive selection of parameters could help when network conditions change during recording. (3) CellReplay’s two-phone setup has some weaknesses. Under mobility, both phones may connect to different base stations and report different performances. Moreover, although we did not observe major interference, greater interference may occur with other providers and conditions. In the future, we will be able to use a single phone with Dual-SIM Dual-Active (DSDA) modem [126] for recording, as DSDA allows simultaneous traffic transmission across two SIMs. Each of these areas represents an opportunity for future improvement. We note, however, that our evaluation results account for the errors caused by these inaccuracies.

Improving CellReplay interpolation accuracy. A straightforward approach is to gather additional data points for interpolation. However, since we need to record data simultaneously (e.g., while walking or driving), we are limited to running only a few workloads with a few phones. Another viable approach is to leverage ML to model complex, workload-dependent network performance and providers’ resource allocation policies. We can train an ML model based on recorded workload and performance traces to predict network performance (e.g., PDOs) for a given test workload. However, this approach may require extensive data collection to capture RAN scheduling behavior, increasing the recording effort and making it time-consuming.

Adding more cellular network specific features. CellReplay could be improved by explicitly emulating cellular network-specific features, especially those that affect application performance. These include radio resource control (RRC) delays, handover, and other relevant factors.

Other limitations. CellReplay probes UDP traffic to record network traces, meaning it cannot capture the effects of network discrimination based on IP protocol types, such as from TCP middlebox intervention [127].

Chapter 6 Discussions and future work

We have outlined several post-thesis future directions specific to DChannel (§3.5), DChannelX (§4.5), and CellReplay (§5.5). This chapter discusses broader post-thesis research opportunities, including the deployment of DChannel and DChannelX, multipath networking over paths with bandwidth–latency trade-offs, and directions that extend beyond cellular network emulation.

6.1 DChannel and DChannelX in today’s 5G deployment

Both DChannel and DChannelX require network operators to expose URLLC to general users through an appropriate interface; however, such an interface has yet to materialize, as URLLC deployment is still ongoing. While URLLC’s current target market is either not yet fully realized (due to ongoing deployment) or remains limited and niche, we show that general mobile applications can already benefit from the low-latency capabilities of URLLC through DChannel and DChannelX.

We propose that network operators can bundle URLLC access with existing network plans. Since DChannel does not require application-level modifications, mobile applications can readily benefit from this service. In this deployment model, operators are responsible for managing the DChannel proxy and deploying the corresponding client software. Alternatively, this responsibility can be offloaded to VPN providers, as DChannel inherently operates as a multipath tunnel. This creates an opportunity for VPN services to offer differentiated value by providing extra low-latency connections over cellular networks, an advantage that is very beneficial for latency-sensitive applications such as mobile gaming.

In contrast, DChannelX does not require active management from network providers, as long as the providers give access to URLLC through an appropriate interface (e.g., via a separate IP address). Instead, the endpoints (client and server) are responsible for scheduling traffic across eMBB and URLLC using a multipath protocol, such as

MPQUIC in DChannelX. We envision DChannelX being particularly useful for emerging applications that are not yet widely deployed, as it requires application-level support for MPQUIC, which currently has limited adoption. This makes DChannelX well suited for distributed XR and cloud XR applications, where both low latency and high bandwidth are essential for correct operation.

6.2 Specialized low latency paths

Future cellular network design. The low latency path or channel in 5G (URLLC) is only equipped with very limited user bandwidth, and hence it is not suitable to serve general application traffic. This limitation arises because the channel is designed to provide both low latency and extremely high reliability (99.999%), resulting in significantly reduced bandwidth. However, general applications do not need the almost-perfect reliability that URLLC guarantees. Both DChannel and DChannelX would also benefit from increased URLLC bandwidth. Future wireless networks (such as 6G) may reconsider this trade-off and provide a low-latency channel with somewhat greater bandwidth and somewhat lower reliability.

Low latency through differentiation. Our insights may apply to other low latency and high bandwidth combinations achieved through service differentiation. A primary example is quality of service (QoS) differentiation, which provides separate latency-optimized and bandwidth-optimized services [128], [129].

The concept of high-bandwidth and low-latency paths can also be applied in networks that do not inherently exhibit such trade-offs. The trade-off itself can be induced by controlling how traffic is distributed across the networks. For instance, we can use cellular and Wi-Fi networks as the low-latency and high-bandwidth channels, respectively. We can treat the cellular network as the low-latency path and send only a small portion of traffic through it; because the queue rarely builds up, this path remains

low-latency. Meanwhile, Wi-Fi can be treated as the high-bandwidth channel, where higher data volumes are sent at the cost of increased latency due to queue buildup.

6.3 Beyond cellular network emulation

Wider-scale measurement study. Our findings on workload and performance dependence behavior in cellular networks motivate a broader measurement study to characterize these dependencies across different operators, network conditions, and locations. An interesting direction for future work is to examine whether different cities exhibit similar dependency patterns, and whether operators in different countries apply distinct network policies. It is also interesting to investigate whether different network conditions (e.g., crowded vs non-crowded) has significantly different dependency patterns. Such measurements could also be crowdsourced and integrated into existing bandwidth testing platforms (e.g., `speedtest.net`).

The resulting dataset could then be used to improve cellular network performance models. These models would also benefit applications, such as for estimating the bandwidth available to a particular workload, which could help congestion controllers make more accurate bandwidth estimations. Additionally, as several prior studies [95], [130], [131] have used network traces to train machine learning models, the collected traces could likewise serve as valuable training data for future ML-based modeling efforts.

Modeling the workload and performance dependency. In CellReplay, we used interpolation between two data points, representing light and heavy workloads, to model the dependency between workload and performance. This model provides a good balance between simplicity and accuracy for our purposes. To achieve a more accurate representation of this dependency, future work could explore testing a wider range of workloads (i.e., sending packet trains of various sizes) and collecting additional data points (e.g., experienced bandwidth or PDOs). We could then apply machine learning

techniques to better capture this relationship and enable predictions of the expected bandwidth for a given workload.

Improving application performance. The workload–performance dependency model can also be leveraged to improve application performance. One primary example is helping applications estimate the actual available bandwidth of the network. We observed that lighter workloads experience lower bandwidth than heavier workloads, which can interfere with reactive application logic. Consider a video streaming application that measures bandwidth to adapt video quality dynamically (§5.4.10). Such an application typically sends an initial frame, measures the bandwidth experienced during that transmission, and then adjusts its subsequent sending rate accordingly. Because lighter workloads (e.g., single frames) experience lower bandwidth, the application underestimates the network’s true capacity and increases its sending rate more slowly. Awareness of this dependency would allow the application to adjust its sending rate faster, resulting in higher video quality.

Furthermore, since heavier workloads tend to achieve higher bandwidth, this knowledge can be exploited to improve data delivery performance. As shown in §5.2.2, for Verizon, the first 50 packets of trains larger than 50 packets were delivered faster than those in a 50-packet train. This observation suggests a potential workaround: applications could append dummy data to their workloads to trigger faster delivery for real data.

References

- [1] S. Sundaresan, N. Feamster, R. Teixeira, and N. Magharei, “Measuring and mitigating web performance bottlenecks in broadband access networks,” in *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2013.
- [2] D. Bhattacharjee et al., “cISP: A Speed-of-Light Internet Service Provider,” in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.
- [3] Y. Einav. “Amazon found every 100ms of latency cost them 1% in sales. ”[Online]. Available: <https://www.gigaspace.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/>
- [4] “Akamai online retail performance report: Milliseconds are critical. ”[Online]. Available: <https://www.ir.akamai.com/news-releases/news-release-details/akamai-online-retail-performance-report-milliseconds-are>
- [5] Q. Jiang et al., “Remotio: Offloading head tracking in an end-to-end xr system,” in *Proceedings of the 16th ACM Multimedia Systems Conference*, 2025, pp. 101–112.
- [6] S. Gao et al., “Xrgo: Design and evaluation of rendering offload for low-power extended reality devices,” in *Proceedings of the 16th ACM Multimedia Systems Conference*, 2025, pp. 124–135.
- [7] E. Cuervo, “Beyond reality: Head-mounted displays for mobile systems researchers,” *GetMobile: Mobile Computing and Communications*, vol. 21, no. 2, 2017.
- [8] M. S. Elbamby, C. Perfecto, M. Bennis, and K. Doppler, “Toward low-latency and ultra-reliable virtual reality,” *IEEE network*, vol. 32, no. 2, pp. 78–84, 2018.

- [9] D. Kanter, “Graphics processing requirements for enabling immersive vr,” *AMD White Paper*, pp. 1–12, 2015.
- [10] V. Agarwal and N. Sastry, ““way back then”: A data-driven view of 25+ years of web evolution,” in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 3471–3479.
- [11] “Valve index headset. ”[Online]. Available:
<https://www.valvesoftware.com/en/index/headset>
- [12] *3GPP Release 16 Description; Summary of Rel-16 Work Items*,
<https://www.3gpp.org/release-16>, [Last accessed on June 16, 2020], Mar. 2020.
- [13] A. E. Gamal, J. Mammen, B. Prabhakar, and D. Shah, “Throughput-delay trade-off in wireless networks,” in *IEEE INFOCOM 2004*, vol. 1, 2004.
- [14] I. 802.1, *Time-Sensitive Networking (TSN) Task Group*,
<https://1.ieee802.org/tsn/> [Last accessed on 27 June 2023], 2023.
- [15] M. K. Vaddiraju et al., “Is wtsn the missing piece for low latency in general-purpose wi-fi?” In *Proceedings of the 26th International Workshop on Mobile Computing Systems and Applications*, 2025, pp. 79–84.
- [16] SpaceX, *Starlink*, <https://www.starlink.com/>, 2023.
- [17] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “Bbr: Congestion-based congestion control,” *Communications of the ACM*, vol. 60, no. 2, pp. 58–66, 2017.
- [18] K. Winstein, A. Sivaraman, and H. Balakrishnan, “Stochastic forecasts achieve high throughput and low delay over cellular networks,” in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL: USENIX Association, Apr. 2013, pp. 459–471, ISBN: 978-1-931971-00-3. [Online]. Available:

<https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/winstein>

- [19] W. Sentosa, B. Chandrasekaran, P. B. Godfrey, H. Hassanieh, and B. Maggs, “DChannel: Accelerating mobile applications with parallel high-bandwidth and low-latency channels,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA: USENIX Association, Apr. 2023, pp. 419–436, ISBN: 978-1-939133-33-5. [Online]. Available: <https://www.usenix.org/conference/nsdi23/presentation/sentosa>
- [20] W. Sentosa, B. Chandrasekaran, P. B. Godfrey, H. Hassanieh, B. Maggs, and A. Singla, “Accelerating mobile applications with parallel high-bandwidth and low-latency channels,” in *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, 2021, pp. 1–7.
- [21] T. Touseef et al., “Boosting application performance using heterogeneous virtual channels: Challenges and opportunities,” in *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, 2023, pp. 139–146.
- [22] W. Sentosa, B. Chandrasekaran, P. B. Godfrey, and H. Hassanieh, “{Cellreplay}: Towards accurate record-and-replay for cellular networks,” in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025, pp. 1169–1186.
- [23] *3GPP Release 15*, <https://www.3gpp.org/release-15>, [Last accessed on May 24, 2020], Apr. 2019.
- [24] 3. G. P. Project, “Study on scenarios and requirements for next generation access technologies,” Tech. Rep., 2017.
- [25] A. Narayanan et al., “A First Look at Commercial 5G Performance on Smartphones,” in *Proceedings of The Web Conference*, 2020.

- [26] *AT&T: 5G Coverage Map*, <https://www.att.com/5g/coverage-map/>, [Last accessed on June 13, 2020], 2020.
- [27] *T-Mobile: The Only Nationwide 5G Network Coverage Map*, <https://www.t-mobile.com/coverage/5g-coverage-map>, [Last accessed on June 13, 2020], 2020.
- [28] *Verizon: 5G Coverage Map*, <https://www.verizon.com/5g/coverage-map/>, [Last accessed on June 13, 2020], 2020.
- [29] H. Hassanieh, O. Abari, M. Rodriguez, M. Abdelghany, D. Katabi, and P. Indyk, “Fast Millimeter Wave Beam Alignment,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.
- [30] E. Sie, X. Wu, H. Guo, and D. Vasisht, “Radarize: Enhancing radar slam with generalizable doppler-based odometry,” in *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications and Services*, 2024, pp. 331–344.
- [31] E. Sie, Z. Liu, and D. Vasisht, “Batmobility: Towards flying without seeing for autonomous drones,” in *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, 2023, pp. 1–16.
- [32] M. Giordani, M. Polese, A. Roy, D. Castor, and M. Zorzi, “A Tutorial on Beam Management for 3GPP NR at mmWave Frequencies,” *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, 2019.
- [33] *MWC: Are Your 5 Fingers Blocking Your 5G?* <https://www.eetimes.com/mwc-are-your-5-fingers-blocking-your-5g/>, [Last accessed on June 24, 2020], Feb. 2018.
- [34] A. Loch, I. Tejado, and J. Widmer, “Potholes Ahead: Impact of Transient Link Blockage on Beam Steering in Practical mm-Wave Systems,” in *The 22nd European Wireless Conference*, May 2016.

- [35] *3GPP TR 38.824 Release 16*, <https://www.3gpp.org/release-16>, [Last accessed on June 16, 2020], Mar. 2019.
- [36] D. Cavalcanti, C. Cordeiro, M. Smith, and A. Regev, “Wifi tsn: Enabling deterministic wireless connectivity over 802.11,” *IEEE Communications Standards Magazine*, vol. 6, no. 4, pp. 22–29, 2022.
- [37] J. Foust, *SpaceX adds laser crosslinks to polar Starlink satellites*, <https://spacenews.com/spacex-adds-laser-crosslinks-to-polar-starlink-satellites/>, 2021.
- [38] E. Ralph, *SpaceX Starlink ‘space lasers’ successfully tested in orbit for the first time*, <https://www.teslarati.com/spacex-starlink-space-lasers-first-orbital-test/>, 2020.
- [39] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. G. Andersen, “SCION: Scalability, Control, and Isolation on Next-Generation Networks,” in *2011 IEEE Symposium on Security and Privacy*, 2011.
- [40] C.-L. Tsao and R. Sivakumar, “On effectively exploiting multiple wireless interfaces in mobile hosts,” in *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009.
- [41] S. G. Hong and C.-J. Su, “ASAP: fast, controllable, and deployable multiple networking system for satellite networks,” in *IEEE Global Communications Conference (GLOBECOM)*, 2015.
- [42] L. Magalhaes and R. Kravets, “Transport level mechanisms for bandwidth aggregation on mobile hosts,” in *Proceedings of the 9th IEEE International Conference on Network Protocols (ICNP)*, 2001.
- [43] H.-Y. Hsieh and R. Sivakumar, “pTCP: An end-to-end transport layer protocol for striped connections,” in *Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP)*, 2002.

- [44] M. Zhang, J. Lai, A. Krishnamurthy, L. L. Peterson, and R. Y. Wang, “A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths,” in *USENIX Annual Technical Conference (ATC)*, 2004.
- [45] J. R. Iyengar, P. D. Amer, and R. Stewart, “Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths,” *IEEE/ACM Transactions on networking (ToN)*, vol. 14, no. 5, 2006.
- [46] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, “Design, implementation and evaluation of congestion control for multipath tcp,” in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.
- [47] Q. De Coninck and O. Bonaventure, “Multipath quic: Design and evaluation,” in *Proceedings of the 13th international conference on emerging networking experiments and technologies*, 2017, pp. 160–166.
- [48] R. Khalili, N. Gast, M. Popovic, and J.-Y. Le Boudec, “Mptcp is not pareto-optimal: Performance issues and a possible solution,” *IEEE/ACM Transactions On Networking*, vol. 21, no. 5, pp. 1651–1665, 2013.
- [49] Q. Peng, A. Walid, J. Hwang, and S. H. Low, “Multipath tcp: Analysis, design, and implementation,” *IEEE/ACM Transactions on networking*, vol. 24, no. 1, pp. 596–609, 2014.
- [50] S. Ferlin, Ö. Alay, O. Mehani, and R. Boreli, “Blest: Blocking estimation-based mptcp scheduler for heterogeneous networks,” in *2016 IFIP networking conference (IFIP networking) and workshops*, IEEE, 2016, pp. 431–439.
- [51] Y. E. Guo, A. Nikraves, Z. M. Mao, F. Qian, and S. Sen, “Accelerating multipath transport through balanced subflow completion,” in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, 2017, pp. 141–153.

- [52] T. Gilad, N. Rozen-Schiff, P. B. Godfrey, C. Raiciu, and M. Schapira, “Mpcc: Online learning multipath transport,” in *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, 2020, pp. 121–135.
- [53] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure, “Exploring mobile/wifi handover with multipath tcp,” in *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*, 2012, pp. 31–36.
- [54] Z. Zheng et al., “Xlink: Qoe-driven multi-path quic transport in large-scale video services,” in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 418–432.
- [55] S. Dhawaskar Sathyanarayana, K. Lee, D. Grunwald, and S. Ha, “Converge: Qoe-driven multipath video conferencing over webrtc,” in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 637–653.
- [56] Y. Zhou et al., “{Augur}: Practical mobile multipath transport service for low tail latency in {real-time} streaming,” in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1901–1916.
- [57] G. Lv et al., “Chorus: Coordinating mobile multipath scheduling and adaptive video streaming,” in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 246–262.
- [58] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann, “Socket intents: Leveraging application awareness for multi-access connectivity,” in *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2013.

- [59] B. D. Higgins et al., “Intentional networking: Opportunistic exploitation of mobile network diversity,” in *Proceedings of the 16th annual international conference on Mobile computing and networking (MobiCom)*, 2010.
- [60] T. Enghardt, P. S. Tiesel, T. Zinner, and A. Feldmann, “Informed access network selection: The benefits of socket intents for web performance,” in *2019 15th International Conference on Network and Service Management (CNSM)*, IEEE, 2019, pp. 1–9.
- [61] T. Enghardt, T. Zinner, and A. Feldmann, “Using informed access network selection to improve http adaptive streaming performance,” in *Proceedings of the 11th ACM Multimedia Systems Conference*, 2020, pp. 126–140.
- [62] M. Z. Shafiq, L. Ji, A. X. Liu, J. Pang, and J. Wang, “Characterizing geospatial dynamics of application usage in a 3G cellular data network,” in *Proceedings IEEE INFOCOM*, 2012.
- [63] D. Xu et al., “Understanding operational 5G: A first measurement study on its coverage, performance and energy consumption,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2020.
- [64] A. Narayanan et al., “A variegated look at 5G in the wild: performance, power, and QoE implications,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2021.
- [65] T. Pauly et al., “An architecture for transport services,” *Internet-Draft draft-ietf-taps-arch-00*, IETF, 2018.
- [66] S. Rosen et al., “Discovering fine-grained RRC state dynamics and performance impacts in cellular networks,” in *Proceedings of the 20th annual international conference on Mobile computing and networking (MobiCom)*, 2014.
- [67] *Multipath TCP in the Linux Kernel v0.94*, <http://www.multipath-tcp.org>, [Last accessed on June 16, 2020], Mar. 2018.

- [68] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, “How Speedy is SPDY?” In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.
- [69] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, “Demystifying page load performance with wprof,” in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.
- [70] M. Halpern, Y. Zhu, and V. J. Reddi, “Mobile cpu’s rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2016, pp. 64–76.
- [71] *3GPP TR 23.725 version 16.2.0 Release 16*, <https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3453>, [Last accessed on January 20, 2022], Jun. 2019.
- [72] R. Netravali et al., “Mahimahi: Accurate record-and-replay for HTTP,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015.
- [73] W. Aqeel, B. Chandrasekaran, A. Feldmann, and B. M. Maggs, “On landing and internal web pages: The strange case of jekyll and hyde in web performance measurement,” in *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020.
- [74] *mitmproxy*, <https://mitmproxy.org>, [Last accessed on May 8, 2024], 2024.
- [75] T. Zimmermann, B. Wolters, O. Hohlfeld, and K. Wehrle, “Is the web ready for http/2 server push?” In *Proceedings of the 14th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2018.
- [76] J. Odvarko. “Har 1.2 spec. ”[Online]. Available: <http://www.softwareishard.com/blog/har-12-spec/>

- [77] M. Ramanujam, H. V. Madhyastha, and R. Netravali, “Marauder: Synergized caching and prefetching for low-risk mobile app acceleration,” in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2021.
- [78] *AndroidViewClient*, <https://github.com/dtmilano/AndroidViewClient>, [Last accessed on April 15, 2022], Mar. 2022.
- [79] *FFmpeg*, <https://ffmpeg.org/>, [Last accessed on April 15, 2022], Jan. 2022.
- [80] *Scenecut extractor*, <https://github.com/slhck/scenecut-extractor>, [Last accessed on April 15, 2022], Dec. 2021.
- [81] S. Gomes, *Resource prioritization – getting the browser to help you*, <https://developers.google.com/web/fundamentals/performance/resource-prioritization>, [Last accessed on June 12, 2020], Jun. 2020.
- [82] *3GPP TS 22.261 version 15.7.0 Release 15*, https://www.etsi.org/deliver/etsi_TS/122200_122299/122261/15.07.00_60/ts_122261v150700p.pdf, [Last accessed on January 20, 2021], Mar. 2019.
- [83] F. Y. Yan et al., “Pantheon: The training ground for internet congestion-control research,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 731–743.
- [84] L. S. Brakmo, S. W. O’Malley, and L. L. Peterson, “Tcp vegas: New techniques for congestion detection and avoidance,” in *Proceedings of the conference on Communications architectures, protocols and applications*, 1994, pp. 24–35.
- [85] M. Dong et al., “Pcc vivace: Online-learning congestion control,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 343–356.
- [86] S. Ha, I. Rhee, and L. Xu, “Cubic: A new tcp-friendly high-speed tcp variant,” *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.

- [87] *Mpshell*,
https://github.com/ravinet/mahimahi/tree/old/mpshell_scripted, [Last accessed on Oct 12, 2025], 2014.
- [88] J. Nejati and A. Balasubramanian, “Wprofx: A fine-grained visualization tool for web page loads,” *Proceedings of the ACM on Human-Computer Interaction*, vol. 4, no. EICS, pp. 1–22, 2020.
- [89] A. Narayanan et al., “A first look at commercial 5g performance on smartphones,” in *Proceedings of The Web Conference 2020*, 2020, pp. 894–905.
- [90] A. Narayanan et al., “Lumos5g: Mapping and predicting commercial mmwave 5g throughput,” in *Proceedings of the ACM Internet Measurement Conference*, 2020, pp. 176–193.
- [91] R. A. Fezeu et al., “An in-depth measurement analysis of 5g mmwave phy latency and its impact on end-to-end delay,” in *International Conference on Passive and Active Network Measurement*, Springer, 2023, pp. 284–312.
- [92] M. Ghoshal et al., “Performance of cellular networks on the wheels,” in *Proceedings of the 2023 ACM on Internet Measurement Conference*, 2023, pp. 678–695.
- [93] NS-3, *NS-3 Network Simulator*, <https://www.nsnam.org/>, [Last accessed: September 20, 2023], 2023.
- [94] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz, “Trace-based mobile network emulation,” in *Proceedings of the ACM SIGCOMM’97 conference on Applications, technologies, architectures, and protocols for computer communication*, 1997, pp. 51–61.
- [95] H. Mao, R. Netravali, and M. Alizadeh, “Neural adaptive video streaming with pensieve,” in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 197–210.

- [96] H. Yeo, Y. Jung, J. Kim, J. Shin, and D. Han, “Neural adaptive content-aware internet video delivery,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, using network trace for video, 2018, pp. 645–661.
- [97] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan, “Vesper: Measuring Time-to-Interactivity for web pages,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA: USENIX Association, Apr. 2018, pp. 217–231, ISBN: 978-1-939133-01-4. [Online]. Available: <https://www.usenix.org/conference/nsdi18/presentation/netravali-vesper>
- [98] P. Goyal, A. Agarwal, R. Netravali, M. Alizadeh, and H. Balakrishnan, “{Abc}: A simple explicit congestion controller for wireless networks,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 353–372.
- [99] D. Ray, J. Kosaian, K. Rashmi, and S. Seshan, “Vantage: Optimizing video upload for time-shifted viewing of social live streams,” in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 380–393.
- [100] S. Abbasloo, C.-Y. Yen, and H. J. Chao, “Wanna make your tcp scheme great for cellular networks? let machines do it for you!” *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 1, pp. 265–279, 2020.
- [101] Y. Xie and K. Jamieson, “Ng-scope: Fine-grained telemetry for nextg cellular networks,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–26, 2022.
- [102] B. Zhang, T. Teixeira, and Y. Reznik, “Performance of low-latency http-based streaming players,” in *Proceedings of the 12th ACM Multimedia Systems Conference*, 2021, pp. 356–362.

- [103] N. Kansal, M. Ramanujam, and R. Netravali, “Alohamora: Reviving HTTP/2 push and preload by adapting policies on the fly,” in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, USENIX Association, Apr. 2021, pp. 269–287, ISBN: 978-1-939133-21-2. [Online]. Available: <https://www.usenix.org/conference/nsdi21/presentation/kansal>
- [104] M. Ramanujam, H. Chen, S. Mardani, and R. Netravali, “Floo: Automatic, lightweight memoization for faster mobile apps,” in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, 2022, pp. 168–182.
- [105] B. Wang, M. Xu, F. Ren, and J. Wu, “Improving robustness of dash against unpredictable network variations,” *IEEE Transactions on Multimedia*, vol. 24, pp. 323–337, 2021.
- [106] D. Yuan, Y. Zhang, W. Zhang, X. Liu, H. Du, and Q. Zheng, “Prior: Deep reinforced adaptive video streaming with attention-based throughput prediction,” in *Proceedings of the 32nd Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2022, pp. 36–42.
- [107] F. Y. Yan et al., “Learning in situ: A randomized experiment in video streaming,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 495–511, ISBN: 978-1-939133-13-7. [Online]. Available: <https://www.usenix.org/conference/nsdi20/presentation/yan>
- [108] S. Wang et al., “Salientvr: Saliency-driven mobile 360-degree video streaming with gaze information,” in *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, 2022, pp. 542–555.

- [109] Y. Ni et al., “Cellfusion: Multipath vehicle-to-cloud video streaming with network coding in the wild,” in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 668–683.
- [110] F. Capozzi, G. Piro, L. A. Grieco, G. Boggia, and P. Camarda, “Downlink packet scheduling in lte cellular networks: Key design issues and a survey,” *IEEE communications surveys & tutorials*, vol. 15, no. 2, pp. 678–700, 2012.
- [111] Y. Xie, F. Yi, and K. Jamieson, “Pbe-cc: Congestion control via endpoint-centric, physical-layer bandwidth measurements,” in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 451–464.
- [112] F. Y. Yan et al., “Pantheon: The training ground for internet congestion-control research,” in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 731–743.
- [113] S. Hemminger et al., “Network emulation with netem,” in *Linux conf au*, vol. 5, 2005, p. 2005.
- [114] L. Rizzo, “Dummynet: A simple approach to the evaluation of network protocols,” *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1997.
- [115] *Network features reference*, <https://developer.chrome.com/docs/devtools/network/reference#throttling-profile>, [Last accessed on May 8, 2024], 2024.
- [116] S. Ashok, S. Tiwari, N. Natarajan, V. N. Padmanabhan, and S. Sellamanickam, “Data-driven network path simulation with ibox,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–26, 2022.

- [117] A. K. Mishra, S. Ayoubi, G. Grassi, and R. Teixeira, “Nemfi: Record-and-replay to emulate wifi,” *ACM SIGCOMM Computer Communication Review*, vol. 51, no. 3, pp. 2–8, 2021.
- [118] M. Claypool, R. Kinicki, M. Li, J. Nichols, and H. Wu, “Inferring queue sizes in access networks by active measurement,” in *Passive and Active Network Measurement: 5th International Workshop, PAM 2004, Antibes Juan-les-Pins, France, April 19-20, 2004. Proceedings 5*, Springer, 2004, pp. 227–236.
- [119] S. C. Chan, K. Chan, K. Liu, and J. Y. Lee, “On queue length and link buffer size estimation in 3g/4g mobile data networks,” *IEEE Transactions on Mobile Computing*, vol. 13, no. 6, pp. 1298–1311, 2013.
- [120] *Pensieve*, <https://github.com/hongzimao/pensieve>, [Last accessed on Sept 17, 2024], 2024.
- [121] B. Taraghi, H. Amirpour, and C. Timmerer, “Multi-codec ultra high definition 8k mpeg-dash dataset,” in *Proceedings of the 13th ACM Multimedia Systems Conference*, 2022, pp. 216–220.
- [122] Y. Rubner, C. Tomasi, and L. J. Guibas, “A metric for distributions with applications to image databases,” in *Sixth international conference on computer vision (IEEE Cat. No. 98CH36271)*, IEEE, 1998, pp. 59–66.
- [123] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, “A control-theoretic approach for dynamic adaptive video streaming over http,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 325–338.
- [124] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson, “A buffer-based approach to rate adaptation: Evidence from a large video streaming service,” in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 187–198.

- [125] A. Hassan et al., “Vivisecting mobility management in 5g cellular networks,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 86–100.
- [126] *Two birds, one stone: Unleashing the full potential for simultaneous 5g cellular connections, thanks to our new qualcomm dsda gen 2 with dual data*, <https://www.qualcomm.com/news/onq/2023/05/unleashing-full-potential-for-simultaneous-5g-cellular-connections-qualcomm-dsda-gen-2-with-dual-data>, [Last accessed on Feb 17, 2025], 2023.
- [127] A. Balasingam et al., “Detecting if lte is the bottleneck with bursttracker,” in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–15.
- [128] J. Babiarz, K. Chan, and F. Baker, “Configuration guidelines for diffserv service classes,” *Network Working Group*, 2006.
- [129] M. Podlesny and S. Gorinsky, “RD network services: differentiation through performance incentives,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2008.
- [130] A. Alomar, P. Hamadani, A. Nasr-Esfahany, A. Agarwal, M. Alizadeh, and D. Shah, “Causalsim: A causal framework for unbiased {trace-driven} simulation,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1115–1147.
- [131] M. Bin-Yahya et al., “Config-snob: Tuning for the best configurations of networking protocol stack,” in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 749–765.