CellReplay: Towards accurate record-and-replay for cellular networks

William Sentosa[†], Balakrishnan Chandrasekaran[‡], P. Brighten Godfrey[†•], Haitham Hassanieh[◇]
[†]*University of Illinois Urbana-Champaign,* [‡]*VU Amsterdam,* [•]*Broadcom,* [◇]*EPFL*

Abstract

The inherent variability of real-world cellular networks makes it hard to evaluate, reproduce, and debug the performance of networked applications running on these networks. A common approach is to record and replay a trace of observed cellular network performance. However, we show that the state-of-the-art record-and-replay technique produces empirically inaccurate results that can cause evaluation bias. This paper presents the design and implementation of 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. The key challenge in achieving high accuracy is to replay varying network behavior in a way that captures its sensitivity to the workload. CellReplay records network behavior under two predefined workloads simultaneously and interpolates upon replay for other workloads. Across various challenging network conditions, our evaluation shows that real-world networked applications (e.g., web browsing or video streaming) running on CellReplay achieve similar performance (e.g., page load time or bitrate selection) to their live network counterparts, with significantly reduced error compared to the prior method.

1 Introduction

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 [12, 13, 22, 23, 34]. 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` [28] 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. [27]. More recently, the Mahimahi network emulator [25] 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., [5,14,18,20,24,26,30,31,36,37,39,41,43,45,46]).

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 §5.6.

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 (§5.9). 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 (§3). 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 workload[1]. 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 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 §6. We release CellReplay alongside with its recorded traces as an open source at https://github.com/williamsentosa95/cellreplay

2 Background and related work

2.1 Cellular network record-and-replay

The goal of record-and-replay network emulation (within the scope of this paper) 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 emu-

[1]To 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. [9]) or a carrier aggregation policy (as observed in [40]) 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.

lated 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 [42] 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 [22]. 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.

2.2 Related work

Network emulators. Popular network emulators, such as NetEm [16] and dummynet [32], can emulate cellular networks. Google Chrome also provides configuration profiles with fixed latency and bandwidth for cellular networks, such as "Fast" and "Slow" 3G [3]. Pantheon [42] 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 [7] extends this by incorporating cross-traffic. However, fixed parameters, by definition, do not capture time-varying effects, which are common in cellular networks.

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 [27]. However, it is designed to emulate WaveLAN, which differs fundamentally from modern cellular networks. More recently, NemFi [21] 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 [25] called Mahimahi, which also included a network emulator derived from CellSim [38] 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.

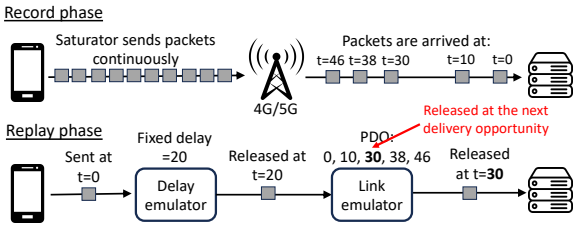

Figure 1: *An overview of Mahimahi's record-and-replay to emulate cellular uplink.*

We detail Mahimahi's record-and-replay approach[2], as it serves as an important reference for this paper. Fig. 1 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 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.

3 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

[2]This 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., [20]). Due to its usefulness, other work has collected newer cellular network traces following CellSim's approach (e.g., [19, 26]) and replayed them in Mahimahi. For simplicity, throughout the rest of this paper, we refer to this record-and-replay method as Mahimahi.

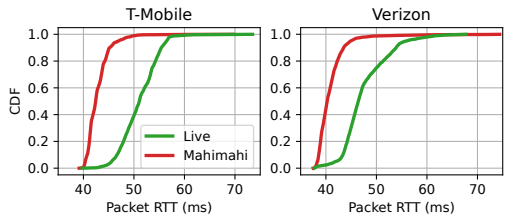

Figure 2: *Packet RTT CDF of Mahimahi emulation and live 5G networks. Mahimahi underestimates the RTT and fails to capture the RTT distribution.*

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).

3.1 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. 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. 1). 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 (§5.3). 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 2 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

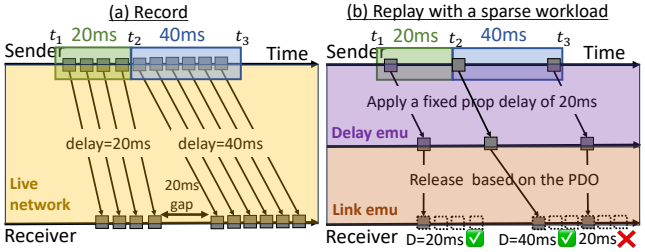

Figure 3: *Mahimahi PDO approach fails to capture the base delay changes for sparse workload.*

propagation delay approach may perform even worse.

This is because PDOs, in principle, only *partially* capture base delay changes. Figure 3 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. 3b, 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.

3.2 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

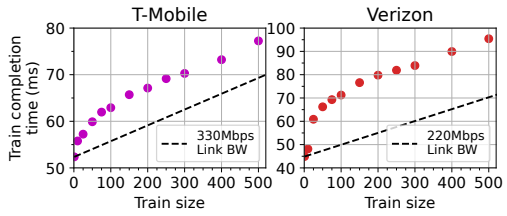

Figure 4: *Mean train completion time (TCT) of different sized trains (N), and TCT if trains were delivered according to the Saturator's observed bandwidth.*

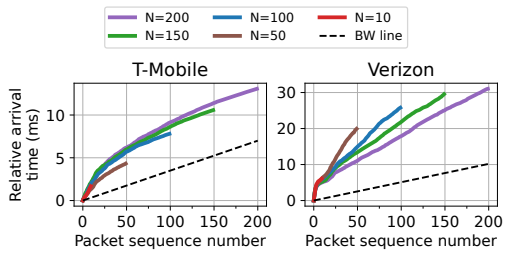

Figure 5: *Mean relative arrival time for each packet of different sized (N) trains.*

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 $(P_0, \ldots, P_{N-1})$. 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 (§5.3). Each test lasted 5 seconds, and there are 12 tests 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 4 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 $P_i \in \{P_0, \ldots, P_{N-1}\}$ 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 5.

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 [9]. This rate-workload dependence was observed across all conditions tested in §5, 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 [12] 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 differ-

ent PDOs, but choosing the right representative workloads remains challenging.

4 CellReplay

4.1 Design overview

At a high level, we want to solve the problems of capturing time-varying base RTT (§3.1) and workload-dependent performance (§3.2) in both record and replay.

We begin with the latter problem (§3.2). 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 (§2.1). Therefore, we can only record a *limited* number of different workloads.

From §3, 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 (§3.1), 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 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

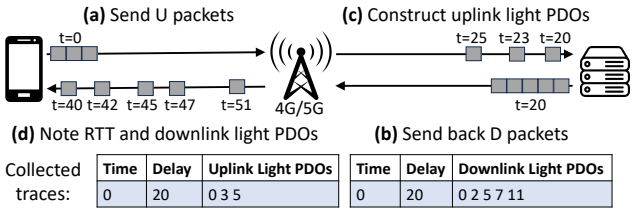

Figure 6: *Packet train workload alongside with its recorded base delay and light PDO traces for uplink and downlink.*

are then used to emulate the network during **replay**. The following subsections detail each of these components: record (§4.2) and replay (§4.3), before returning to calibration (§4.4), which is best understood after seeing the rest of the design.

4.2 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 6 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

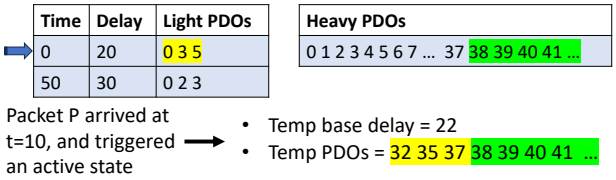

Figure 7: *Temporary base delay and PDOs constructed when a CellReplay first enters an active state.*

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 [38]. 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 [38] 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.

4.3 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 (§4.4), 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 7. CellReplay searches for the most recent base 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 §3.2, 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 state[3]. 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 dequeueing. 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 trigger the procedure to reenter the active state, as described above.

Table 1: *Parameters set in calibration phase (§4.4).*

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
$comp(s)$	Delay compensation for s-byte packets
B	Bottleneck buffer size in bytes

4.4 Parameter Calibration

We describe how to select values for the parameters in Table 1. The parameters U, D, G_{min}, F, and $comp(s)$ 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 [11]. For details, see §A.3.

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 §3.2) using a fixed train gap that is conservatively large enough to ensure the link returns to its light-workload state.

[3]Replaying 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 §3.1 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.

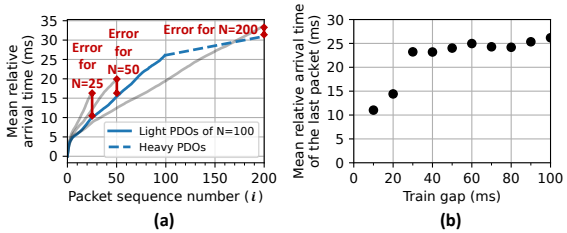

Figure 8: *(a) Estimating error if a 100-sized train is used to estimate {25, 50, 200}-sized trains. (b) G_{min} is chosen as the smallest train gap that results in performance similar to a large train gap (i.e.,* 30 ms*).*

In our implementation, we use a 100 ms gap, and the set of train sizes we consider is $\{5, 25, 50, 75, \ldots, X\}$ 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 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. 8(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, $g \in 100, 90, 80, ..., 10$ 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. 8(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 $r_{last}(G_{min})$. For details, see §A.2.

Inferring *comp*(). We profile how RTT is affected by packet size by sending randomly sized packets between {100, 200, ..., 1400} bytes every 50 ms to a receiver that responds with a 100-byte ACK. We then measure the RTT difference for a packet size of s compared to the RTT of 1400-byte packets and model this difference as $comp(s)$. We describe this in more detail in §A.1.

5 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 [25]. We implemented CellReplay record tool in Java and Python 3 to send and receive UDP packets. We extended the Mahimahi shell to support CellReplay replay, allowing unmodified applications to run inside the shell and experience the emulated network conditions induced by CellReplay. For more details, refer to §B.

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). For full details on the environments and their calibration parameters, refer to §C. Finally, we present a use case of using CellReplay and Mahimahi to evaluate ABR algorithms.

5.1 Experimental setup

We designed two test setups: a *live network* and an *emulation* test setup, as shown in Figure 9. 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

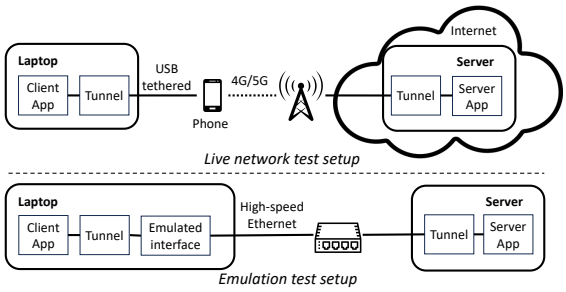

Figure 9: *Live network and emulation application test setup.*

to 5G or 4G 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 [42]). 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.

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.

5.2 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 [6]. 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 internal pages corresponding to the 10^{th}, 30^{th}, 50^{th}, 70^{th}, and 90^{th} 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. Refer to §D for the exact list of pages and their page composition).

We utilized Mahimahi's HTTP record-and-replay framework [25] 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` [2], 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 [47] extension).

In our test setup (Fig. 9), 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 [29], 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 [4, 20], 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 [35]) 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.

5.3 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) [33], 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 ($\approx$ 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 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 §3.1 and §3.2) 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 (as well as tests) lasted for 10 s. 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 10). 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 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.

5.5 Web browsing test

We evaluated CellReplay's accuracy for the web browsing by fetching both landing and internal pages (listed in Table 3) 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 11). 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 §5.6). 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 §5.6), 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 behav-

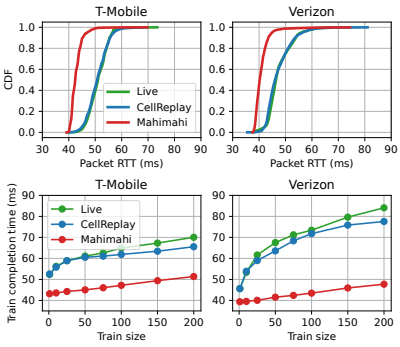

Figure 10: *The CDF of RTTs (above) and mean TCTs (below) from the packet RTT and packet train tests, respectively, run on both live network and emulations.*

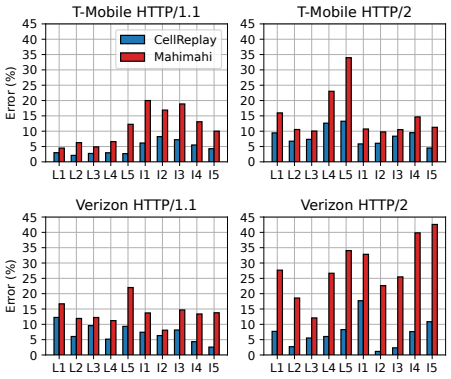

Figure 11: *Emulation distribution error across different web-page load tests with HTTP/1.1 and HTTP/2.*

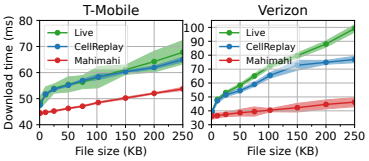

Figure 12: *Mean small file download times along with its 95% CI.*

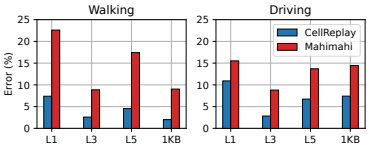

Figure 13: *Emulation distribution error of three web PLTs and 1KB file download time under mobility.*

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

5.6 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 12 shows the file download times for sizes between 1 KB and 250 KB. Consistent with the observation in §5.4, 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 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. 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. Refer to §E.3 for the full results, including mean file download times for 1 MB and 10 MB files and their respective distribution emulation errors.

5.7 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 §5.5 and §5.6, and reported the average error across both operators.

Figure 14 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

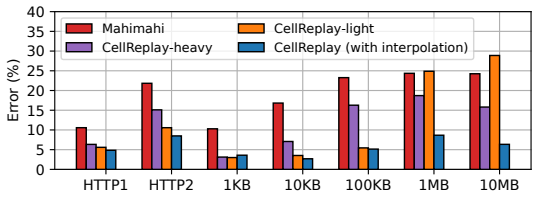

Figure 14: *Emulation distribution error of CellReplay with and without light and heavy PDOs interpolation.*

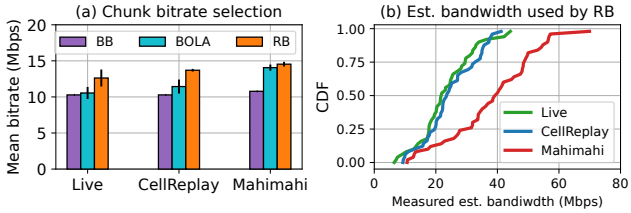

Figure 15: *(a) Mean ABR bitrate selection result. Mahimahi shows positive bias to BOLA. (b) Reported estimated bandwidth from RB.*

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.

5.8 Mobility and other network conditions

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 (§5.2), 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 total, we spent 1.67 h walking and 2.45 h driving around the loop.

Figure 13 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 capture[4]. We indeed confirmed that `driving` triggered at least two handovers[5], 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, as indicated by the shape of the CDF curve, which is closer to the live network's curve compared to Mahimahi. Refer to §E.2 for the CDF curve for 1KB download and L3 page load times.

Additionally, we tested CellReplay in a basement and a crowded library. In both conditions, CellReplay produced significantly lower error compared to Mahimahi, with an emulation distribution error of 5.74% (vs. 15.22%) in the basement and 8.47% (vs. 22.51%) in the crowded library. Refer to E.1 for more details.

5.9 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, 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 algorithms[6] from [4] 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 [17] 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 15a 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

[4]This 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.

[5]There 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.

[6]This setup also includes MPC [44] and Pensieve [20], but these were not tested, as MPC requires modifying the hardcoded MPC table, and Pensieve requires retraining.

results suggest that 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 15b, 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.

6 Discussion and future work

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

Use cases. As shown in §5.9, 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 [22]. However, packet drops are more frequent during handovers in mobility [15]. (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 [1] 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 [8].

7 Conclusion

This paper exposes the difficulty of accurate record-and-replay emulation and presents CellReplay, which more faithfully captures real-world cellular network performance characteristics. CellReplay's approach of dual-workload recording and interpolated replay provides the community with a more accurate platform for evaluating research in cellular environments. We also hope this work inspires the community to explore future designs that can make record-and-replay emulation even more faithful to live deployments.

Acknowledgements: We sincerely thank our shepherd, Kyle Jamieson, and the anonymous reviewers for their valuable suggestions. We also thank Keith Winstein for helpful discussions on Mahimahi and CellSim. Additionally, we thank Qinjun Jiang, Sam Yuan, and Pradnyan Khodke, who helped drive the car while the author conducted experiments. This project was supported by gifts from T-Mobile and Cisco, and a grant from the IBM-Illinois Discovery Accelerator Institute.

References

[1] 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, 2023. [Last accessed on Feb 17, 2025].

[2] mitmproxy. https://mitmproxy.org, 2024. [Last accessed on May 8, 2024].

[3] Network features reference. https://developer.chrome.com/docs/devtools/network/reference#throttling-profile, 2024. [Last accessed on May 8, 2024].

[4] Pensieve. https://github.com/hongzimao/pensieve, 2024. [Last accessed on Sept 17, 2024].

[5] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. Wanna make your tcp scheme great for cellular networks? let machines do it for you! *IEEE Journal on Selected Areas in Communications*, 39(1):265–279, 2020.

[6] Waqar Aqeel, Balakrishnan Chandrasekaran, Anja Feldmann, and Bruce 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.

[7] Sachin Ashok, Shubham Tiwari, Nagarajan Natarajan, Venkata N Padmanabhan, and Sundararajan Sellamanickam. Data-driven network path simulation with ibox. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–26, 2022.

[8] Arjun Balasingam, Manu Bansal, Rakesh Misra, Kanthi Nagaraj, Rahul Tandra, Sachin Katti, and Aaron Schulman. Detecting if lte is the bottleneck with bursttracker. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2019.

[9] Francesco Capozzi, Giuseppe Piro, Luigi Alfredo Grieco, Gennaro Boggia, and Pietro Camarda. Downlink packet scheduling in lte cellular networks: Key design issues and a survey. *IEEE communications surveys & tutorials*, 15(2):678–700, 2012.

[10] Stanley CF Chan, KM Chan, Ke Liu, and Jack YB Lee. On queue length and link buffer size estimation in 3g/4g mobile data networks. *IEEE Transactions on Mobile Computing*, 13(6):1298–1311, 2013.

[11] Mark Claypool, Robert Kinicki, Mingzhe Li, James Nichols, and Huahui 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*, pages 227–236. Springer, 2004.

[12] Rostand AK Fezeu, Eman Ramadan, Wei Ye, Benjamin Minneci, Jack Xie, Arvind Narayanan, Ahmad Hassan, Feng Qian, Zhi-Li Zhang, Jaideep Chandrashekar, 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*, pages 284–312. Springer, 2023.

[13] Moinak Ghoshal, Imran Khan, Z Jonny Kong, Phuc Dinh, Jiayi Meng, Y Charlie Hu, and Dimitrios Koutsonikolas. Performance of cellular networks on the wheels. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, pages 678–695, 2023.

[14] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. {ABC}: A simple explicit congestion controller for wireless networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 353–372, 2020.

[15] Ahmad Hassan, Arvind Narayanan, Anlan Zhang, Wei Ye, Ruiyang Zhu, Shuowei Jin, Jason Carpenter, Z Morley Mao, Feng Qian, and Zhi-Li Zhang. Vivisecting mobility management in 5g cellular networks. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 86–100, 2022.

[16] Stephen Hemminger et al. Network emulation with netem. In *Linux conf au*, volume 5, page 2005, 2005.

[17] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 187–198, 2014.

[18] Nikhil Kansal, Murali Ramanujam, and Ravi 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)*, pages 269–287. USENIX Association, April 2021.

[19] Gerui Lv, Qinghua Wu, Yanmei Liu, Zhenyu Li, Qingyue Tan, Furong Yang, Wentao Chen, Yunfei Ma, Hongyu Guo, Ying Chen, et al. Chorus: Coordinating mobile multipath scheduling and adaptive video streaming. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pages 246–262, 2024.

[20] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 197–210, 2017.

[21] Abhishek Kumar Mishra, Sara Ayoubi, Giulio Grassi, and Renata Teixeira. Nemfi: Record-and-replay to emulate wifi. *ACM SIGCOMM Computer Communication Review*, 51(3):2–8, 2021.

[22] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. A first look at commercial 5g performance on smartphones. In *Proceedings of The Web Conference 2020*, pages 894–905, 2020.

[23] Arvind Narayanan, Eman Ramadan, Rishabh Mehta, Xinyue Hu, Qingxu Liu, Rostand AK Fezeu, Udhaya Kumar Dayalan, Saurabh Verma, Peiqi Ji, Tao Li, et al. Lumos5g: Mapping and predicting commercial mmwave 5g throughput. In *Proceedings of the ACM Internet Measurement Conference*, pages 176–193, 2020.

[24] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. Vesper: Measuring Time-to-Interactivity for web pages. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 217–231, Renton, WA, April 2018. USENIX Association.

[25] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: accurate {Record-and-Replay} for {HTTP}. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 417–429, 2015.

[26] Yunzhe Ni, Zhilong Zheng, Xianshang Lin, Fengyu Gao, Xuan Zeng, Yirui Liu, Tao Xu, Hua Wang, Zhidong Zhang, Senlang Du, et al. Cellfusion: Multipath vehicle-to-cloud video streaming with network coding in the wild. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 668–683, 2023.

[27] Brian D Noble, Mahadev Satyanarayanan, Giao T Nguyen, and Randy H Katz. Trace-based mobile network emulation. In *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 51–61, 1997.

[28] NS-3. NS-3 Network Simulator. https://www.nsnam.org/, 2023. [Last accessed: September 20, 2023].

[29] Jan Odvarko. Har 1.2 spec, 2007.

[30] Murali Ramanujam, Helen Chen, Shaghayegh Mardani, and Ravi Netravali. Floo: automatic, lightweight memoization for faster mobile apps. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 168–182, 2022.

[31] Devdeep Ray, Jack Kosaian, KV Rashmi, and Srinivasan Seshan. Vantage: optimizing video upload for time-shifted viewing of social live streams. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 380–393. 2019.

[32] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

[33] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. A metric for distributions with applications to image databases. In *Sixth international conference on computer vision (IEEE Cat. No. 98CH36271)*, pages 59–66. IEEE, 1998.

[34] William Sentosa, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Haitham Hassanieh, and Bruce 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)*, pages 419–436, Boston, MA, April 2023. USENIX Association.

[35] Babak Taraghi, Hadi Amirpour, and Christian Timmerer. Multi-codec ultra high definition 8k mpeg-dash dataset. In *Proceedings of the 13th ACM Multimedia Systems Conference*, pages 216–220, 2022.

[36] Bo Wang, Mingwei Xu, Fengyuan Ren, and Jianping Wu. Improving robustness of dash against unpredictable network variations. *IEEE Transactions on Multimedia*, 24:323–337, 2021.

[37] Shibo Wang, Shusen Yang, Hailiang Li, Xiaodan Zhang, Chen Zhou, Chenren Xu, Feng Qian, Nanbin Wang, and Zongben Xu. Salientvr: saliency-driven mobile 360-degree video streaming with gaze information. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 542–555, 2022.

[38] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 459–471, Lombard, IL, April 2013. USENIX Association.

[39] Yaxiong Xie and Kyle Jamieson. Ng-scope: Fine-grained telemetry for nextg cellular networks. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–26, 2022.

[40] Yaxiong Xie, Fan Yi, and Kyle 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, pages 451–464, 2020.

[41] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association.

[42] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, 2018.

[43] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 645–661, 2018. using network trace for video.

[44] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno 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*, pages 325–338, 2015.

[45] Danfu Yuan, Yuanhong Zhang, Weizhan Zhang, Xuncheng Liu, Haipeng Du, and Qinghua 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*, pages 36–42, 2022.

[46] Bo Zhang, Thiago Teixeira, and Yuriy Reznik. Performance of low-latency http-based streaming players. In *Proceedings of the 12th ACM Multimedia Systems Conference*, pages 356–362, 2021.

[47] Torsten Zimmermann, Benedikt Wolters, Oliver Hohlfeld, and Klaus 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.

A Calibration details

A.1 Inferring $comp()$: derivation

We examine the impact of packet size on RTT by sending packets with sizes sampled uniformly at random from {100, 200, ..., 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.

A.2 Inferring F: derivation

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 8b, 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).

A.3 Inferring B.

We use a classical max-min approach [11] 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 is calculated as $(RTT_{max} - RTT_{min}) \cdot C$, where C is the observed link capacity [10].

B 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

Table 2: *All tested conditions that includes stationary, walking, and driving scenarios. We tested T-Mobile (TM) and Verizon (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=25, 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 was with the user driving in a loop around the university area	TM	5G MB	U=10, D=75, G=100, F=5

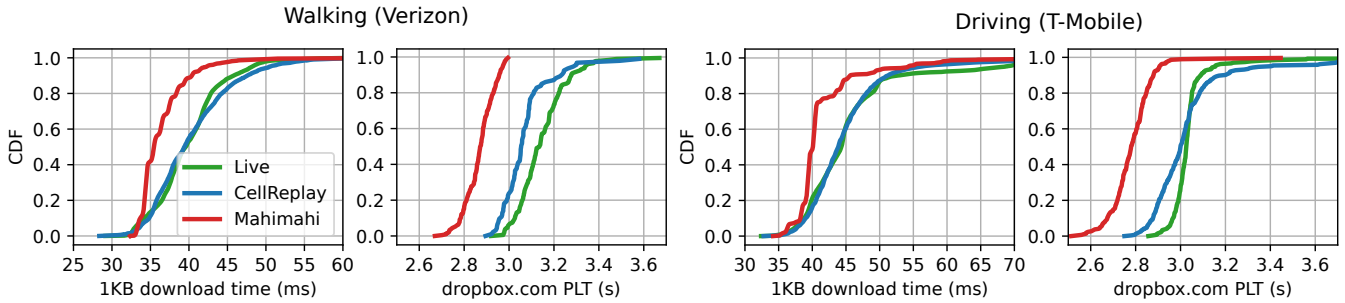

Figure 16: *CellReplay manages to capture more application performance variability under mobility compared to Mahimahi. We cut the CDF graph as the tail is too long.*

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.

C Experimental conditions

The environments used for evaluation are listed in Table 2.

D Tested web pages list

The list of web pages used for testing and its details on page composition is detailed in Table 3

E More evaluation results

E.1 Non-ideal 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 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

Table 3: *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.*
Landing	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
Internal	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

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 17, across the two network conditions and

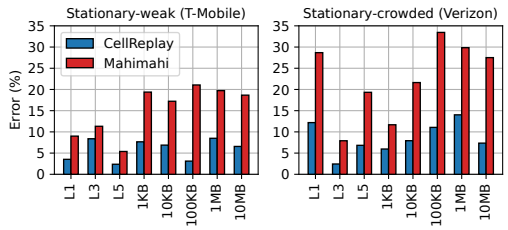

Figure 17: *The web PLT and file download emulation distribution error when testing under non-ideal stationary cases.*

Table 4: *Mean file download time (in ms) of medium-sized files with its emulation distribution error.*

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

providers, CellReplay has an average distribution error of 5.96% for web-page loads and 7.9% for file downloads. 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.

E.2 Experiments under mobility

Figure 16 demonstrates CellReplay's ability to capture application performance under mobility scenarios more accurately than Mahimahi.

E.3 Medium-sized file download test

Table 4 shows the mean file download time alongside its distribution emulation error for CellReplay and Mahimahi, compared to the live networks.

F Ethics

We note that CellReplay cannot be used to collect other users' packets; it only collects traces of its own user. Moreover, traces collected by CellReplay do not contain any private information. Hence, we believe our work does not raise any ethical concerns.