Public Review for
NemFi: Record-and-replay to emulate WiFi

Abhishek kumar Mishra, Sara Ayoubi, Giulio Grassi, Renata Teixeira

Reproducibility of experiments in a WiFi context is a known challenge. In this work, the authors create a trace-driven WiFi emulator called NemFi that allows modeling of transmission opportunities of uplink and downlink directions, packet loss, frame aggregation, and media access control behavior. The latter two concepts are unique for WiFi when compared with other similar tools that have been built for cellular networks. A small-scale evaluation is performed with a client, server, and one access point for the three different applications of iperf, scp, and DASH.

Public review written by
Joseph Camp
Southern Methodist University

NemFi: Record-and-replay to emulate WiFi

Abhishek kumar Mishra
Inria Saclay
abhishek.mishra@inria.fr

Sara Ayoubi
Nokia Bell Labs
sara.ayoubi@nokia-bell-labs.com

Giulio Grassi
Inria Paris
giulio.grassi@inria.fr

Renata Teixeira
Inria Paris
renata.teixeira@inria.fr

ABSTRACT

This paper presents *NemFi*: a trace-driven WiFi emulator. *NemFi* is a record-and-replay emulator that captures traces representing real WiFi conditions, and later replay these traces to reproduce the same conditions. In this paper, we demonstrate that the state-of-the-art emulator that was developed for cellular links cannot emulate WiFi conditions. We identify the three key differences that must be addressed to enable accurate WiFi record-and-replay: WiFi packet losses, medium-access control, and frame aggregation. We then extend the existing cellular network emulator to support WiFi record-and-replay. We evaluate the performance of *NemFi* via repeated experimentation across different WiFi conditions and for three different types of applications: speed-test, file download, and video streaming. Our experimental results demonstrate that average application performance over *NemFi* and real WiFi links is similar (with less than 3% difference).

CCS CONCEPTS

• **Networks** → **Network measurement**; **Network performance analysis**; **Network experimentation**;

KEYWORDS

WiFi, trace-driven emulation, record & replay tools

1 INTRODUCTION

WiFi is increasingly more popular due to the widespread use of mobile devices (*e.g.* smartphones, laptops, tablets, smartwatches, etc.)[7]. The quality of WiFi connectivity varies drastically from place to place and over time due to several factors such as poor network configuration, old equipment, fluctuating demands of users, congestion, and coverage. As many of today's applications and services will be running over WiFi, it is crucial to evaluate the performance of these applications in different network conditions. The variability of WiFi makes it hard to predict how an application/service will work with just a few experiments. Testing in one/few settings tells little of how a service will behave when deployed at a large scale over a long period.

There are different options for evaluating networked applications and services before deployment: simulation, testbed experiments, and emulation. Simulation is the easiest way to experiment with different wireless network conditions. Network simulation tools (*e.g.* NS-2 [3], NS-3 [1], OMNET++ [9], to name a few) are used to mimic the behavior of wireless networks in a software-based environment. The advantages of simulation are repeatability, control, configurability, and scalability. The main limitation of simulation tools, however, is that they require the user to tune the different parameters, *e.g.* level of interference, congestion, loss rate (among others), which may not reflect real wireless network conditions. Even with good parameter settings, a simulator cannot capture the complex inter-dependencies of real systems.

At the other end of the spectrum, there is testbed experimentation, where developers evaluate their applications over deployed wireless links either over testbeds or by relying on volunteer testers. The results of such experiments capture the impact of real wireless network conditions. The major disadvantage of experimentation is that it offers no repeatability, and is difficult to scale. The variability of wireless networks makes it hard to reproduce results. The results of experimentation are, therefore, hard to interpret and one cannot distinguish the issues with application versus wireless conditions.

Finally, trace-driven emulation [6, 10] involves recording traces in deployed wireless networks and later replay these traces to reproduce the recorded network conditions. The clear benefits of trace-driven emulations are its ability to capture real network conditions and the repeatability of the experiments. One can run the same network conditions several times, which eases application or system debugging, and enables comparative analysis of different applications or protocols over the same network conditions.

While there exist trace-driven emulators for cellular [10] and HTTP traffic [6], to the best of our knowledge, there exists no such solution for WiFi. Adapting cellular emulation for WiFI is not trivial due to the many fundamental differences between WiFi and cellular, in particular, in how they manage access to the shared medium, how they react to packet loss, and other technology-specific protocols. These differences makes the existing cellular network emulator unable to accurately emulate WiFi.

Motivated by the advantages of trace-driven emulation, and the lack of such a tool for WiFi, we design and implement *NemFi*:[1] a trace-driven emulator for WiFi. *NemFi* extends the state-of-the-art cellular emulator [10] to support accurate WiFi emulation. We begin by identifying the challenges of trace-driven emulation for WiFi. These challenges help us identify the key design decisions of *NemFi*'s record-and-replay modules. We demonstrate through extensive evaluations that *NemFi* accurately emulates WiFi in various network conditions.

The main contributions of this paper are therefore:

- We identify the challenges of trace-driven emulation for WiFi.

[1] *NemFi*'s source code, and network traces: https://gitlab.inria.fr/mabhishe/nemfi

- We introduce a novel trace-driven emulator for WiFi, which makes it possible to evaluate network applications and services over emulated WiFi conditions.

The rest of this paper is organized as follows. In Section 2, we provide a brief overview of the existing trace-driven cellular emulator, followed by the list of challenges for designing a trace-driven emulator for WiFi. In Section 3, we explain the design of *NemFi*. In Section 4, we validate the accuracy of *NemFi*'s design via a series of experiments for various types of applications and mobility scenarios. We conclude the manuscript in Section 5 and provide future research directions.

2 BACKGROUND AND MOTIVATION

This section first introduces the record-and-replay emulator for cellular networks developed by Winstein *et al.* [10] and then discusses the challenges to adapt this emulator to WiFi.

2.1 Trace-Driven Emulation for Wireless Networks

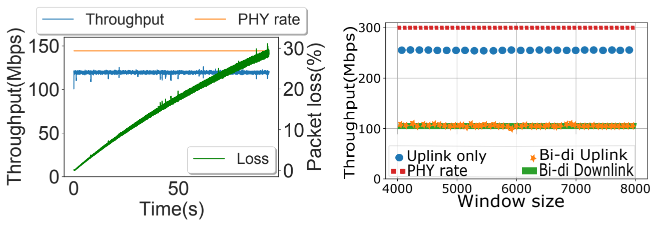
Winstein *et al.* [10] introduced a cellular network emulator to evaluate their new transport protocol for low-latency high-throughput transmission over wireless cellular networks. Their proposed emulator consists of two main modules: a record module, known as the *Saturator*, designed to record a trace of cellular network variability. This trace is then passed to the replay module, *CellSim*, which replays the trace to reproduce the captured network conditions. Concretely, the Saturator is a software module running on two end-hosts connected to each other via a cellular interface. During record, the Saturator aims to saturate the uplink and downlink channels by pushing MTU-sized UDP packets and recording the time each packet is received on the other end. These timestamps, also referred to as "delivery opportunities", represent the time MTU-sized packets were able to effectively cross the cellular link (in each direction). During replay, Cellsim runs on a PC connected to two communicating end-hosts via Ethernet. CellSim listens for each incoming packet (in either direction), consults the trace of delivery opportunities, and delay packets accordingly to match the time packets were effectively delivered during the record.

The emulator proposed by Winstein *et al.* has been specifically designed to emulate cellular networks, and hence cannot accurately record and replay WiFi variability. In particular, in cellular networks, there are per device queues. If the bottleneck is the cellular network, then the congestion at the base-station is mostly self-induced and the effect of cross-traffic is muted. Moreover, in cellular networks, the uplink and downlink communications take place on different time slices and do not interfere with each other. In WiFi, on the other hand, the medium is shared and hence delivery opportunities are shared between the upstream and the downstream flows as well as with competing traffic. Further, LTE base-stations hold much larger queues than WiFi and allow for more re-transmissions. In WiFi, the queues are smaller and packet losses more common.

2.2 Challenges of WiFi Network Emulation

The design of a trace-driven emulator for WiFi brings a number of challenges.

Capturing WiFi delivery opportunities: As we have mentioned



(a) (b)

Figure 1: a) Throughput and packet loss observed by Saturator for the static client, and b) Fair-share between the uplink and the downlink in WiFi

in the previous section, the Saturator captures cellular variability by saturating the uplink and the downlink simultaneously. While this approach works for cellular, we cannot adopt the same approach for WiFi. This is due to the difference in how cellular and WiFi manage access to the shared medium. In cellular different carrier frequency bands are dedicated for the uplink and downlink transmission [5]. However, in WiFi, the transmitting nodes contend for access to the shared medium. This means that if we saturate the uplink and downlink simultaneously, we capture the delivery opportunities on the upstream under the contention from the downstream transmission, and vice-versa. This is problematic in the case where the captured trace is used to evaluate the performance of applications that mostly push traffic in a single direction.

Identifying the saturation point: To capture the delivery opportunities, it is important to saturate the available channel capacity. The cellular emulator achieves this by employing a large window of packets-in-flight. Saturator adjusts the window size to keep the observed RTT between 750 milliseconds and 3 seconds. Using such a large window size ensures that the Saturator has a persistent queue of packets at the bottleneck link, which in return means that it is effectively saturating the link. Further, by setting a threshold on the window-size, the Saturator ensures that it will not overflow the network queue and thus induce packet loss. Adopting such a large threshold is possible in cellular because cellular employs large queues to deal with rapidly changing network variability and multi-second outages [10]. However, we cannot employ a similar approach in WiFi for two main reasons: (1) WiFi employs a much smaller queue than cellular, 2) the maximum bitrate available for a WiFi client (*i.e.* PHY rate) is not fixed and is affected by channel conditions and WiFi losses. The latter means that to effectively capture the available bandwidth in WiFi we need to dynamically adjust the window size to adapt to the changing PHY rates. Failure to do so, may cause the record to overflow the network queue and thereby induce packet losses, which could potentially cause the WiFi network to further reduce the PHY rate. Figure 1a demonstrates the effect of using a larger static threshold on the observed packet loss rate in WiFi. These results were obtained by running the Saturator on an end-device connected to another end-host via WiFi in a controlled setup with ideal WiFi conditions (no competing traffic, end-host in close proximity to the WiFi access-point). In Figure 1a, we observe that while Saturator has already reached saturation point (steady-state) under one second, the packet losses keep increasing and reach 30% of all packets sent within 90 seconds. Note that due to the overhead of control packets Saturator's throughput is only 85% of the PHY rate.

Capturing WiFi losses: WiFi losses are more common in practice than cellular, because WiFi employs a smaller packet buffer and a fewer number of L2-retransmissions. To accurately reproduce WiFi variability, it is important to capture and replay WiFi losses. Failure to do so will make all recorded WiFi traces appear lossless. This will impact the performance of applications replayed over these traces as the effect of WiFi losses is absent. The importance of capturing WiFi losses further highlights the need to avoid inducing packet loss during the record as it raises the challenge of isolating losses due to buffer overflow from WiFi losses.

Emulating WiFi-specific features: It is essential to account for other WiFi-specific features like frame aggregation as the client is capable of achieving a much higher link utilization by sending frames in batches. This feature is at the heart of improvements in recent WiFi standards like IEEE 802.11 n, ac, and ax. Frame aggregation could be A-MSDU or A-MPDU with varying parameters adopted by devices. Hence, we require a solution that does consider these factors.

3 *NEMFI* : DESIGN AND IMPLEMENTATION

We begin by providing a brief overview of *NemFi*'s system design. *NemFi* extends the state-of-the-art trace-driven emulator for cellular to support WiFi emulation. *NemFi* records WiFi packet delivery opportunities in a trace that can be later replayed to emulate the recorded WiFi conditions. To achieve this, *NemFi* is designed with two main components: a record module and a replay module. *NemFi*'s record module records WiFi network variability by capturing the time MTU-sized packets were able to effectively cross the WiFi link, as well as the time packets were dropped due to WiFi losses. These traces of packet deliveries and packet losses are then passed to the replay module. *NemFi*'s replay module is built on top of Mahimahi's [6], a framework designed to record-and-replay HTTP traffic. Similar to MahiMahi's replay module, *NemFi* is built as a Unix shell. When an application is running inside the shell, all of its incoming and outgoing packets will be intercepted and placed inside a queue. These packets will be first delayed for a fixed amount of time to emulate one-way propagation delays, and then the packet-delivery and packet-loss traces will be inspected to determine the fate of each packet. Either the packet will be dropped because it coincides with a packet loss event, or the packet will be released. In the latter case, the packet will be released from the queue to match the recorded packet-delivery trace.

CellSim and MahiMahi's replayshell are identical in how they replay the packet-delivery trace, however, the key difference is that MahiMahi is built using Unix shells, whereas CellSim requires to run on a dedicated machine connected to both the client and the server during replay. For this reason we opted to use MahiMahi's replayShell, since it's easier to use in practice and requires less hardware for the replay.

3.1 *NemFi*'s record

The goal of the record is to capture the variability of a WiFi link over time. To achieve this, *NemFi*'s record module runs on two machines: a sender and a receiver (as illustrated in Figure 2). The sender is connected to the receiver via two links: the WiFi link being measured and a reliable link for feedback. A separate reliable link provides a quick feedback loop to allow our record tool to quickly adapt its sending rate, and avoid over-filling the network queue. As mentioned in Section 2.2, the key challenges of designing a record mechanism for WiFi are: (1) How do we accurately capture the delivery opportunities on the uplink and downlink simultaneously given the effect of contention? (2) How do identify when we have saturated the channel? Recall that identifying the saturation point is key to prevent introducing packet losses, which may cause WiFi to reduce the sending rate. In the remainder of this section, we explain how we modify the Saturator to address these two concerns.

3.1.1 Capturing WiFi delivery opportunities: Given the limitations of saturating the WiFi uplink and downlink simultaneously, we modified the Saturator to run the saturation in a single direction only. The intuition behind this decision is that by saturating in a single direction, we omit the effect of contention and are able to capture all the delivery opportunities available on the WiFi link overtime. This trace provides us with all the information we need to then emulate the WiFi link accurately for any type of application or service (regardless of their transmission mode). The next question is how to allocate the delivery opportunities between the uplink and downlink during replay. To answer this question, we conducted a set of experiments to understand how bandwidth is distributed between uplink and downlink in WiFi. Figure 1b illustrate our results. We ran the Saturator while varying the window size between 4000 to 8000 MTUs, and for each window size value, we run the Saturator to saturate the channel in both directions (denoted as Bi-di in Figure 1b) or a single direction (uplink only). Our results show that in all cases there is a fair share of delivery opportunities between the uplink and the downlink streams. Further, we observe that the throughput we achieve while saturating in a single direction matches the sum of throughput in each direction. This result indicates that by recording the delivery opportunities in a single direction, we are able to then emulate any combinations of uplink-downlink transmissions by employing fair resource sharing during replay.

3.1.2 Identifying Saturation point: The next challenge that we need to address is that of identifying the saturation point, which is represented in WiFi by the PHY rate. The PHY rate indicates the maximal bitrate available between the wireless client and the access point. The issue is that this value is dynamic since WiFi adapts the PHY rate based on the wireless network conditions. Hence to address the problem of dynamic saturation point we need to measure the PHY rate overtime and quickly adapt the Saturator's sending rate to match the PHY rate as it changes. We measure the PHY rate by extracting station dump information from the client WiFi driver using Linux utility *iw* every 25 ms. With repeated experimentation, we found that this frequency strikes a good balance between getting a good estimate of the maximum bitrate overtime without inducing too much run-time overhead, due to continuous routine calls to the driver, which could interfere with the saturation.

To adapt the window size to match the PHY rate overtime, we equip *NemFi* with a rate control algorithm, presented in Algorithm 1.

We start by setting the window size to 5. *NemFi*'s rate control algorithm is then periodically invoked with the last PHY rate reading and the current Saturator's sending rate as input. The goal of the rate-control algorithm is to decide whether to increase or decrease the window size, and by how much, to match the latest PHY rate value. To achieve this, we follow an approach similar to proportional–integral–derivative controller that is widely used in industrial control systems [4]. We set the last PHY rate value as the "theoretical bound". Next, we compute the difference between the Saturator's current throughput and the theoretical bound, and denote this value as the 'error' *i.e.* how far we are from the theoretical bound. The error is then used to determine by how much we need to increase the window size The window size is $error * \alpha$, where α is a constant that controls how aggressively we approach the PHY rate. We set α as $\frac{theoreticalBound}{1500}$. Our intuition behind setting α proportional to the current PHY rate is that the higher the PHY rate, the larger the window size increments should be to quickly converge to the saturation point. Further, our empirical results have shown that by setting the denominator to 1500, we strike a good balance between quickly converging to the PHY rate while minimizing the risk of overshooting.

Algorithm 1 NemFi's Rate Control algorithm

```
1:  procedure SatControl($currentUplinkPHY$, $currentThroughput$)    ▷ Input feedback variables
2:      $theoreticalBound \leftarrow currentUplinkPHY$
3:      $error \leftarrow theoreticalBound - currentThroughput$
4:      $diffThroughptGain \leftarrow lastThroughput - currentThroughput$
5:      $satFlag \leftarrow 0$
    ▷ Check if we need window size adaptation
6:      if $satFlag \neq 1 \quad \& \quad diffThroughptGain \neq 0$ then
7:          $window \leftarrow window + alpha * error$
8:      else
9:          $satFlag \leftarrow 1$
10:     end if
    ▷ Decrease window on a drop in PHY rate
11:     if $lastTheoreticalBound > theoreticalBound$ then
12:         $window \leftarrow 0.8 * window$
13:         $satFlag \leftarrow 0$
14:     else if $lastTheoreticalBound < theoreticalBound$ then
15:         $satFlag \leftarrow 0$
16:     end if
17: end procedure
```

Next, we need to figure out if we reached the saturation point. To achieve this, we define the differential throughput gain as the difference between the Saturator's current and last throughput value. While the differential gain is positive, we keep increasing the window. Once the differential gain becomes zero, it means we reached the saturation point, and we set the saturation flag to true to avoid increasing the window size any further.

The final task of the rate control algorithm is to adapt the window size in the event of PHY rate change. When we observe a drop in PHY rate due to deteriorating channel conditions, we decrease the window size to 80% and turn the saturation flag to false. It causes the algorithm to get to a new saturation state by re-adapting the window size. An aggressive drop in the window size would result in more transient time in reaching the next saturation state. We observe by repeated experiments that with 80% drop, we are reasonably quick in getting to the next saturation state even in extremely mobile scenarios. Similarly, we assign the saturation flag as false on noticing an improvement in PHY rate to increase the window size further.

At the end of the record phase, we obtain a trace where each entry comprises of: the timestamp and the sequence number of successfully sent packets, the throughput and the percent loss rate at that instant along with the PHY rate and the window size. For the replay, we process the recorded-trace to give a new trace as input. The replay-trace entries comprise of: the delivery opportunity, the throughput, the sequence number, and the percent loss rate. We covert the timestamps in recorded trace to a time-series in milliseconds from 0 to the duration of the record, which serves as packet delivery opportunities during the replay.

3.2 *NemFi*'s replay

The goal of *NemFi*'s replay tool is to retrace the same time-based conditions recorded in the trace with packet deliveries and packet losses. There are three main challenges that we need to address for replaying WiFi: 1) Sharing opportunities between the uplink and the downlink, 2) Replaying WiFi losses, and 3) Emulating frame aggregation.

Firstly, we need to share delivery opportunities between the uplink and the downlink flows as they communicate on a shared spectrum in WiFi. This is not present in MahiMahi's linkshell as it is designed for cellular, where uplink and downlink traffic are allocated to different time slices. Hence, we introduce a weighted round-robin like approach to share the delivery opportunities between the uplink and the downlink. Concretely, we define *percentShare* to represent the uplink share of the medium; where 1 - *percentShare*[2] represents the downlink share. At the time of the next delivery opportunity, *NemFi* generates a random number between 0 and 1. If the number is smaller than 1 - *percentShare* and the downlink queue is not empty, the downlink queue seizes the delivery opportunity. If the dowlink queue is empty, and the uplink queue is not, the uplink queue seizes the delivery opportunity. The process repeats similarly for all subsequent delivery opportunities. To emulate propagation delays, we introduce a constant delay of half the round-trip time on both the uplink and downlink packet queues.

The next challenge is to emulate WiFi losses. We use the instantaneous loss rate at the current index of the packet delivery trace when a packet is being read from the socket into the corresponding packet queue. If the loss rate turns out to be positive, we generate a random number between 0 and 1. If this number is less than the current loss rate, then we drop the packet read from the socket and that packet is not appended to either of the packet queues.

Finally, to emulate frame aggregation, we adopt the model introduced by da Hora *et al.* [2]. Concretely, we estimate the total number of aggregated frames at different PHY rates that we observe

[2]In our evaluation, we set this value to 0.5 (as per the results in Figure 1b) to achieve fair share between the uplink and downlink

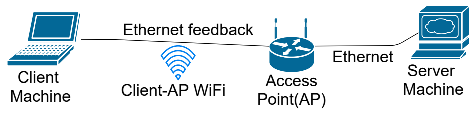


Figure 2: Experimental setup for NemFi

in each entry of the packet-delivery trace. Using the model [2], we estimate the frame aggregation per PHY rate for the specific AP and stations used in the experimental setup(Section 4.1). This gives us an inference of the number of packets to be grouped together for sending it to the output queue. We additionally observe that all packets which are part of the same aggregate have practically the same delivery timestamps. We do not group the whole set of delivery opportunities when we have missing sequence numbers, due to the losses in the aggregated frame during the record phase. On detecting the missing sequence number in the input trace, we do not group more packets in the same emulation opportunity.

4 EVALUATION

In this section we first describe the experimental setup used to evaluate *NemFi*, and then we detail the analysis of *NemFi*'s validation: first, in terms of recording with fidelity the WiFi link capacity fluctuations and losses, then in terms of replaying a real application on the recorded traces.

4.1 Experimental setup

To validate *NemFi* we deploy an in-lab testbed where we capture and replay WiFi traces in a controlled environment (see Figure 2).

We have *NemFi* running on two machines, a client and a server. The client machine is connected to a WiFi access point via a wireless card and an Ethernet cable. The wireless connection is used to monitor the wireless link and capture traces, while the cable connection is used to exchange control messages with the server, connected via cable to the same access point. The client sends UDP packets to the server via the wireless link (saturating the channel) and receives acknowledgements from the server via Ethernet.

In our evaluation, the client and server are HP Elitebook 840 G2 running 5^{th} Generation Intel Core i7-5600U 2.6 GHz (max turbo frequency 3.2-GHz) with 8 GB RAM and 512 GB SSD hard drive. Both machines have Intel I218LM Gigabit network connection (10/100/1000 NIC) cards. In addition, the client is equipped with an Intel Wireless 7265 with *iwlwifi* driver and A-MPDU frame aggregation enabled. The access point is a TP-Link AC1750, Archer C7 V5 which supports IEEE 802.11ac/n/a on 5GHz and IEEE 802.11b/g/n on 2.4GHz.

For the evaluation, we conduct our experiments using IEEE 802.11ac/n/a on a 5GHz channel with 20 MHz width, and we consider three main scenarios: (1) *Ideal* with stationary client and server machines that are 2 meters apart. (2) *Static and far* scenario, where the machines are stationary and the client is 12 meters from the access point. (3) *Mobile* scenario, with the client machine moving indoors to replicate home/office mobility patterns.

For the replay, we use the same setup described above, and evaluate *NemFi* using 3 applications with different data-transfer patterns. We use Iperf [8] to simulate applications with TCP-upload type of traffic. SCP for TCP-download type of traffic (client machine downloading a large file from the server). Finally, DASH is representative of a more complex data exchange pattern, where bursts of traffic is downloaded followed by periods of zero-traffic, and the application adapt its bitrate based on the network throughput. In this case, an error in the record-replay may lead to a different behaviour of the application, with different choices for the video bitrate to download. For this scenario, the client machine downloads a one minute video from the server machine encoded in several bitrates, from 248 Kbps to 52 Mbps.

4.2 Validation of *NemFi*'s record

In this section we validate the trace record component of *NemFi*. In particular, we verify that: (1) the instantaneous throughput reported in the trace is consistent with the WiFi PHY rates, (2) *NemFi* does not induce additional losses, and (3) *NemFi* fully utilises the link capacity. For the latter, we use the model defined by da Hora [2] to estimate the instantaneous wireless link capacity given the PHY rate and report the ratio (in percentage) of the theoretical link capacity and the throughput reported by *NemFi*.

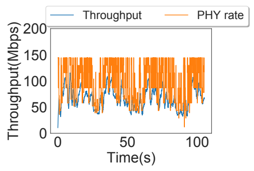
We report the instantaneous recorded throughput and link utilization in the most challenging case, the *Mobile* scenario, where we expect rapid fluctuations in the supported PHY rates. Figure 3a illustrates the time series of the captured throughput and the PHY rate in one example experiment. We see that *NemFi* throughput closely tracks the wireless PHY rate and quickly adapts to changes.

Figure 3b reports the ratio between the *theoretical link capacity* and the throughput measured by *NemFi*. Theoretical link capacity here refers to the maximum throughput that could be achieved at a particular PHY rate by the client. We deduce this capacity using the model of da Hora *et al.* [2]. We obtain the ratio by using mean throughput and theoretical link capacities observed in 100 ms bins. The figure shows that the average link utilization is always close to 100 percent for all the encountered physical rates. For some PHY rates like 115 Mbit/s, a slightly lower link utilization is expected, since they are recorded only for extremely short duration.

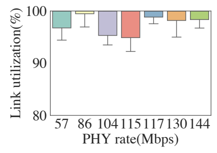
We study packet loss in the *Ideal* scenario where the wireless loss rate is expected to be close to zero. The ideal setting allows us to detect if *NemFi* introduces packet loss by over-estimating the channel capacity; whereas in other scenarios, it is difficult to distinguish between WiFi losses and *NemFi* induced losses. We infer packet losses from the gaps in the sequence numbers of packets received at the server. Figure 3c reports the packet loss seen across all the experiments in the *Ideal* case. We conclude that *NemFi* does not lead to an increase in packet loss, since the measured loss rate remains at a negligible value of 0.2% on average, in contrast with the results reported in Figure 1a when we ran the Saturator in the same setting.

4.3 Validation of *NemFi*'s replay

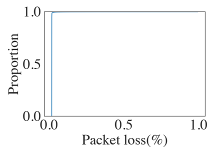
In this section we validate *NemFi*'s replay phase, by showing that the performance of a diverse set of applications in *NemFi*'s emulated environment matches the performance of the same applications in a real environment. To do so, we design the following experiment: for each scenario and application, we run *NemFi* to record the WiFi trace, and, when this phase is complete, we run the application in



(a) Example comparing NemFi throughput with PHY rate in Mobile scenario.

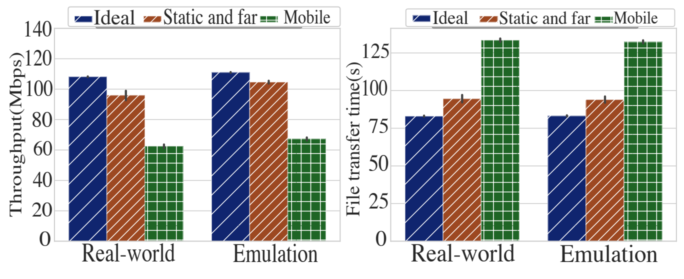


(b) Percentage link utilization achieved during the record in Mobile scenario.

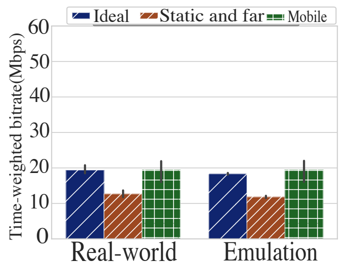


(c) Packet loss rates during NemFi's record in ideal scenario.

Figure 3: Evaluation of *NemFi's* record.



(a) **(b)**



(c)

Figure 4: *NemFi* vs real-world WiFi experiments for (a) Iperf, (b) SCP, and (c) DASH with varying client conditions

the real environment (without using *NemFi*). We later replay the same application on the recorded traces. In this case, we use the same setup as in Figure 2, but the client machine uses *NemFi* over the Ethernet cable instead of the WiFi to communicate with the server machine. We repeat this process 15 times for each case and compare the average performance of each application across runs in the same setting, in the real case and with *NemFi* emulation.

The applications we chose are quite different, thus we use different metrics to evaluate the accuracy of *NemFi* in replaying the WiFi conditions. For *Iperf* we consider the average throughput measured by the server per experiment, while for *SCP* we measure the total file transfer time in each run. For DASH, we consider the time-weighted average of bitrates of video chunks during a one-minute video to capture the video streaming performance of an experimental run.

Figure 4 presents the performance of each application, in the different scenarios, for the real and emulated environment. In Figure 4a, we see that the average throughput achieved by Iperf in the *Ideal* condition for real-world experiments over WiFi and *NemFi*'s replay is within 2%. Even in *Static and far* and *Mobile* scenarios, we replay Iperf throughput with an accuracy of 93%. For SCP (Figure 4b), we observe that the file transfer time with emulation differs by at most 2% from the real-environment experiments under the same conditions.

Finally, in Figure 4c we observe that the difference in the time-weighted bit-rates for WiFi and *NemFi*'s replay in *Ideal* and *Static and far* scenarios is less than 3%. In the *Mobile* scenario, where WiFi link capacity fluctuates more because of client mobility, *NemFi*'s emulated environment leads to a difference in the DASH performance close to 2%.

5 CONCLUSION & FUTURE REMARKS

In this paper, we introduced *NemFi* a novel trace-driven emulator for WiFi. We identified a number of challenges that need to be addressed to develop an accurate record-and-replay tool for WiFi, and we demonstrate how *NemFi* addressed these challenges. Our evaluations show that *NemFi* accurately captures WiFi conditions by capturing the variability in delivery opportunities and WiFi losses. We hope that by releasing *NemFi* others will be able to build on our work both by improving and further evaluating *NemFi* as well as by using it to evaluate networked systems over emulated WiFi. In particular, our evaluation focused on indoor scenarios with limited mobility, hence further evaluation is required for using *NemFi* in other scenarios.

[1] Prakash Agrawal and Mythili Vutukuru. 2016. Trace based application layer modeling in ns-3. In *2016 Twenty Second National Conference on Communication (NCC)*. IEEE, 1–6.

[2] Diego Neves da Hora, Karel Van Doorselaer, Koen Van Oost, Renata Teixeira, and Christophe Diot. 2016. Passive wi-fi link capacity estimation on commodity access points. In *Traffic Monitoring and Analysis Workshop (TMA) 2016.*

[3] Teerawat Issariyakul and Ekram Hossain. 2009. Introduction to network simulator 2 (NS2). In *Introduction to network simulator NS2*. Springer, 1–18.

[4] Michael A Johnson and Mohammad H Moradi. 2005. *PID control.* Springer.

[5] Swarun Kumar, Ezzeldin Hamed, Dina Katabi, and Li Erran Li. 2014. LTE radio analytics made easy and accessible. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 211–222.

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

[7] S. O'Dea. 2020. Number of smartphone users worldwide from 2014 to 2020 (in billions). *URL https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/* (2020).

[8] Ajay Tirumala. 1999. Iperf: The TCP/UDP bandwidth measurement tool. *http://dast. nlanr. net/Projects/Iperf/* (1999).

[9] Andras Varga. 2010. OMNeT++. In *Modeling and tools for network simulation*. Springer, 35–59.

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