Painting the DragonEye: From Inanimate Data to Interactive Control Emulation of Cellular Networks

Jingchu Liu*, Bhaskar Krishnamachari†, Sheng Zhou*, Zhisheng Niu*
*Tsinghua National Laboratory for Information Science and Technology,
Tsinghua University, Beijing, China
†University of Southern California, Los Angeles, CA, USA
liu-jc12@mails.tsinghua.edu.cn, bkrishna@usc.edu,
{sheng.zhou, niuzhs}@tsinghua.edu.cn

ABSTRACT

Data from mobile cellular networks has been extremely valuable in helping us understand traffic dynamics, monitor network operations, and conceive better deployment plans. Among these applications, trace-driven emulation is particularly interesting due to its potential in enabling open, reproducible, and cost-effective examination of innovative designs of cellular networks. Nevertheless, the inanimate nature of off-line data restricts it from being directly used for testing interactive network control algorithms, which may change the original data distribution. To address this issue, we present the DragonEye interactive emulation framework. DragonEye combines offline network data with microscopic refining and reacting models to emulate the live interaction between mobile users and the cellular network. For demonstration, we implement DragonEye using session-level user traffic logs captured from a real WLAN. This implementation is then used to test a reinforcement-learning-based base station (BS) sleeping control algorithm. Results show that emulated users can interactively queue and cancel requests in response to dynamic sleeping operations, which demonstrates the effectiveness of DragonEye in emulating live interaction using offline data.

Keywords

Network Emulation, data-driven networking, software-defined networking, reinforcement learning

1. INTRODUCTION

"I fear to draw their eyes. They will come alive and fly away." - Chinese fairy tale "Drawing the Dragon Eye".

Mobile cellular networks are not only data pipes but also valuable data sources themselves. Network measurements and signaling logs have been widely used to monitor operations and diagnose problems in face of anomaly [15]. Mobile traffic records also served as insightful lenses for observing human behavior at the population scale [3]. Further more, network data can be incorporated back into the planing and operations of cellular networks for application such as remote channel estimation [8], hand-off prediction [6], etc.

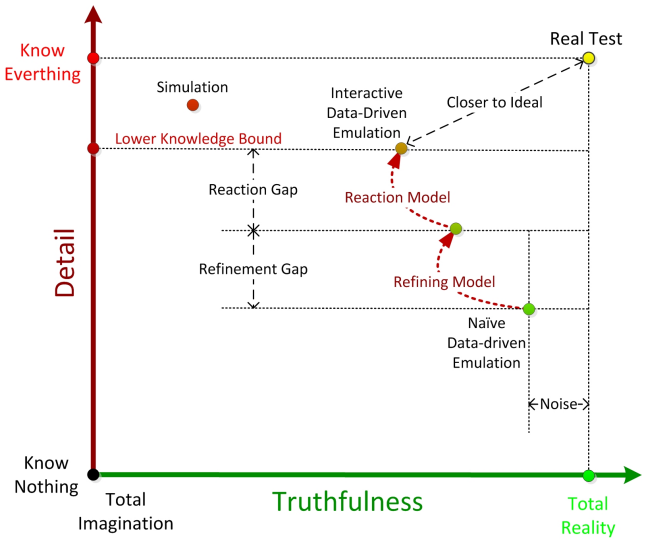

Figure 1: Comparison among different testing methods and illustration for the refinement and reaction gaps for data-driven emulation.

Amongst such applications, data-driven network testing is particularly intriguing due to its favorable balance between fidelity and complexity. Data-driven network testing tries to closely imitate the behavior of a real network by modulating an emulator with data captured from the target network. To put the advantages of this method into perspective, we compare it with other testing methods in Fig. 1. A real-life testbed (top-right corner) is both genuine and fully-detailed. It does not include artificial assumptions that will undermine the truthfulness of evaluation, and it can provide sufficiently detailed knowledge for any algorithms being evaluated. But since it can be quite expensive to build and and complex and time-consuming to operate, it is often only the last step in the evaluation of algorithms. In comparison, model-based simulation (top-left corner) is relatively cheaper and more tractable and is therefore a popular alternative. The ample choice of analytical models also guarantees flexible trade-

offs between simplicity and fidelity. Nevertheless, an artificial model may deviate from reality due to the computational intractability of accurately modeling real-life and our incomplete understanding of the modeled system. In contrast, data-driven emulation (center) strikes a better balance among the conflicting constraints. Data is the truthful projection of real world as we record them. This means a data-driven emulator can honestly reproduce the network state and context at the time of capture. Also, real-time data manipulation for data-driven emulation has become feasible and efficient enough with the help of advanced data processing software. Both advantages can potentially improve the openness, reproducibility, and cost-effectiveness in innovative research on cellular networks.

However, there are two major gaps that must be addressed in the realization of data-driven testing. The first gap is the **reaction gap**. The distribution of network data depends on the network state at the moment of capture. Thus any network operations that may change the internal network state are also in danger of altering the subsequent distribution of data. Examples of such dynamic operations include base station (BS) sleeping which may cause packet delay and loss, load balancing which may change the interference and congestion level, scheduling that may invoke user-side flow control, and etc. Reaction to the operations above will cause the network trace to deviate from the captured one. Insisting on driving the emulator with the original trace is likely to result in biased results. The other gap is the **refinement gap**. Limited by our data capturing and storage capability, network data is often coarse-grained, only providing summaries or aggregate statistics about the complete network process. Datasets like this may not even meet the minimum knowledge requirements of certain control algorithms and thus unable to drive tests for them. Note in cases where captured trace is fine-grained enough, the refinement gap diminishes.

DragonEye mitigates the gaps above by introducing microscopic **traffic refining** and **interactive reacting** models into the data-driven modulation process. The traffic refining model generates fine-grained traffic information from the coarse-grained dataset. For example, the captured dataset may only tell how many requests a user sent in total. By imposing a model for the request generation process, we can extend our knowledge to the exact issuing time of each request. The interactive reacting model maintains internal states of the network and governs the state transition under dynamic interactions. For example, if a BS choose to queue a user request, the model will tell us whether the user will wait or simply give up. With these two models, network data can be reused to test algorithms which may cause a network process realization that is different from the captured one. Indeed, the emulation will become partially synthetic after introducing the two models. This is equivalent to sacrificing truthfulness for more detailed knowledge, and will cause the final system to drift left-wards while moving up in the quadrature of Fig. 1. But it is important to note that modeling microscopic behavior is often easier than macroscopic behavior. Microscopic models will also preserve the dominating influence of the captured data. Hence, data-driven emulation is likely to be closer to real-life testbed than pure model-based simulation provided that the captured data is of proper quality.

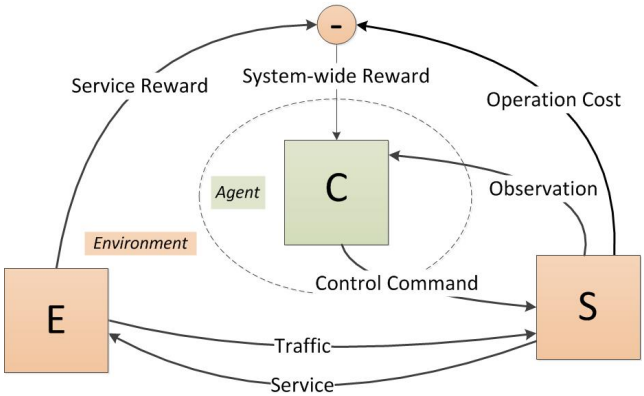

Figure 2: Components of DragonEye and their interaction. E: Traffic emulator, S: traffic server, C: controller.

The rest of this article is organized as follows. In Section 2 we describe the structure and operation of the DragonEye framework. In Section 3 we show how to implement DragonEye using data collected from a campus wireless local-area network (WLAN). The implemented emulator is then tested against a BS sleeping algorithm in Section 4. Related work is reviewed in Section 5. And the paper is concluded in Section 6.

2. FRAMEWORK OVERVIEW

2.1 System Structure

DragonEye consists of three components: a traffic emulator (E) which imitates the behavior of a collection of traffic sources such as mobile devices, a traffic server (S) which serves as the data plane of the network and processes traffic, as well as a controller (C) which is the control plane of the network and directs the operation of the data plane. The interaction among these components is illustrated in Fig. 2: E generates traffic for S, which receives the traffic and provide certain services under the (full or partial) direction of C. To issue commands wisely, C makes observations through S about the traffic and system states. Depending on the services provided, the received traffic may be accepted, delayed, or denied. E and S will then emit a scalar metric to depict the quality of interaction: E emits a scalar to measure the service quality while S a scalar to quantify the cost of services. These rewards are summarized and passed back to C, which then evaluates its current policy and improves it towards higher rewards.

2.2 Interfaces and Operation Flow

We define a set of interface methods for the three components in DragonEye. Actual implementation should materialize these interfaces irrespective of the system under test and the data used. Their invocation flow during emulation is shown in Fig. 3. Emulation starts with E. At the start of each epoch, the `generate_traffic()` method of E is called to get a detailed description of the generated traffic. If no more traffic can be generated, the emulation will terminate; otherwise, the generated traffic is fed to S as the argument

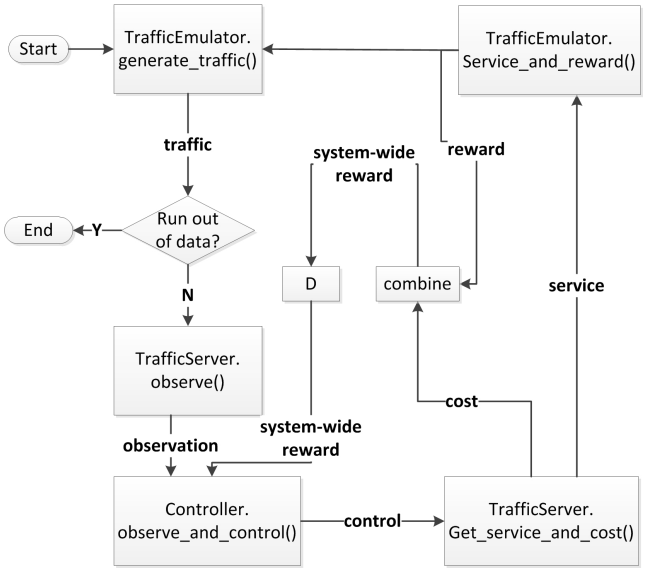

Figure 3: Flow graph for invoking interface methods during emulation. The D module delays the input by one epoch.

of `observe(traffic)`. Inside this method, S summarizes the current traffic information and system states and return the summary as an observation vector. The observation vector is then passed on to C as the first argument of the `observe_and_control(observation,last_reward)` method. C will decide the control command under current observation based on certain operation policy. The control command is fed back to S through `get_service_and_cost(control)`, in which S processes the traffic and prepares a service description for E. At the same time, it will also quantify how expensive the services are using a scalar cost. E receives the service description in `serve_and_reward(service)`. It modifies its internal states based on the services provided and emits a scalar to quantify its satisfaction level. The service reward and operational cost are combined and buffered for the next epoch as the second argument of `observe_and_control( observation, last_reward)`. C can use the combined reward to evaluate and improve its current policy.

2.3 Integrating Reinforcement Learning

Our data-driven emulator is designed to enable open, reproducible, and cost-effective testing of network control algorithms. In this regard, we have to clarify how control algorithms can be integrated into this system. Here we are interested in the general task of learning the optimal control policy. Reinforcement learning [14] is an effective framework for tackling this task. A reinforcement learning agent can learn an optimal policy that maximizes the accumulated rewards by continuously interacting with an environment and gradually improving its policy. To apply reinforcement learning, we must first define the agent, the environment, and their interfaces. The correspondence between the reinforcement learning agent and the controller C is quite obvious, as they both serve the role of learning through trial and error. Meanwhile, the environment is not a single component in our system. Instead, it is the collection of E, S, as well as the reward combiner as shown in Fig. 2.

Column Name	Explanation	Example
uid	Unique ID for each user	41117355
bldn	Name of the building.	Dining Hall 1
start	Session start time (Unix).	1409500812697
dur	Session duration (ms).	295551
dmns	Domain list.	[a.com, b.net]
prvdr	Provider of each domain.	[Tencent, Apple]
types	Type of each domain.	[Portal, Shopping]
bytes	bytes/domain.	[8500, 341]
reqs	HTTP requests/domain.	[5, 1]

Table 1: Explanation and example for dataset record columns.

3. IMPLEMENTATION EXAMPLE

In this section, we implement[1] the proposed DragonEye framework using a realistic WLAN dataset. We will describe the dataset, the goal and gaps towards data-driven emulation, and the refining and reacting models used.

3.1 Dataset Description

The dataset we use is captured from a campus WLAN from September 2014 through January 2015. It contains the session-level HTTP traffic summaries of around 20,000 users. Here "session" is defined as the period in which a user generates HTTP requests and pauses for less than 5 minutes each time. If the user pauses longer, the subsequent requests will be summarized into the following session.

Each record summarizes the per-domain HTTP activity of each user during each session with the following fields: session ID, user ID, building name[2], start time (UNIX time), duration (milli-seconds), list of the domains requested, the corresponding provider, domain type, as well as the total number of HTTP requests and bytes requested at each domain. Explanations and examples for these columns are summarized in Table 1.

3.2 Emulation Goal and the Gaps

Goal: Traffic in wireless networks is known to be highly nonuniform in both space and time. As a result, most BSs are under-utilized for a large portion of the time [13]. To avoid wasting energy in such situations, BS sleeping operations turn down under-utilized BSs in low-traffic periods and wake them up when the traffic becomes heavy again. However, user traffic issued during sleeping periods will need to be either queued or dropped, causing delay or loss respectively. This raises the fundamental energy-delay/loss tradeoff of BS sleeping operations. Existing literature has studied this problem using analytical models [16]. But the models used are generally limited, such as Poisson models, and may not closely approximate the true dynamics of realistic traffic. The goal of data-driven emulation in this case is to study different sleeping policies in realistic traffic conditions.

[1]Implementation is hosted on Github: `https://github.com/zaxliu/dqn4wirelesscontrol/tree/master/sleep_control`

[2]More granular knowledge of user location (access point (AP) level) is hidden to preserve user privacy.

Refinement gap: the minimum knowledge detail requirement for this particular application is knowing the exact issuing time, location, and meta-info, e.g. bytes and target domain, of each user request. However, since such fine-grained information is aggregated into session summaries in the dataset, the exact information of each individual requests is unknown to us. Another problem arises with spatial aggregation. The exact access point (AP) location of each session is aggregated to the physical-building level for privacy reasons. Therefore we do not know the true association between user sessions and APs.

Reaction gap: the dynamic nature of BS sleeping operations is quite obvious. Networking service will be unavailable during sleeping periods, therefore the user requests that are issued then will have to be either queued or dropped. On the user side, the flow control and retransmission mechanisms are likely to be invoked in response, causing the transmission schedule of subsequent requests to deviate from the one captured in the dataset. On the network side, the queuing of new requests may also influence how the BS schedule subsequent transmissions.

3.3 Refining and Reacting Models

To mitigate the gaps above, we assume the following refining and reacting models. Besides, we also define the rewarding mechanism to reflect the trade-off between energy consumption and delay/cost.

Virtual large cell: to cope with the uncertainty of user location at sub-building level, we assume all the users in a physical building are served by a large-coverage BS instead of multiple small-coverage APs. This assumption is inspired by the fact that the actual geographical layout of the buildings involved are rather far apart and the size of a single building matches the coverage size of a typical cell in cellular communication systems.

Byte and epoch allocation: we assume a uniform byte and epoch allocation model to translate coarse session-level summary to fine-grained request description[3]. According to this model, each byte in a session belongs to a request in that session with equal probability, and each request belongs to an epoch within that session with equal probability too. This essentially results in a multinomial byte-per-request and request-per-epoch probability distribution.

User and network state machines: we model the reaction of users and the network using state machines. At the user side, the transmission of each request follows the state machine shown in Fig. 4. All requests are initialized as "pending" and remain there before being transmitted. If a request should be sent out in a particular epoch, its state is modified to "waiting" and start transition according to the service received: if this request is successfully served, its state is then changed to "served" and transition is terminated; if otherwise it is queued, its state remains at "waiting"; if it is rejected from service, its state will be switched back to "pending" with probability p and re-allocated to one of the epochs left in this session with uniform probability, or tagged as "failed" with probability $1 - p$ and terminate

[3]Note that any other distribution could also be used.

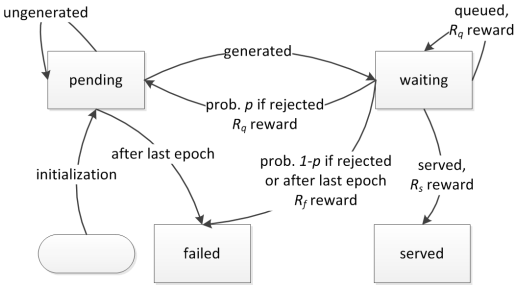

Figure 4: Request generation state machine

further transition. All "pending" and "waiting" requests are also tagged as "failed" after the last epoch of a session.

The network-side state machine is relatively simple. In each epoch, S first puts all incoming requests in a request queue. If the control command received is "serve", then all queued requests will be served and queue cleared; otherwise if the command is "queue", the queued requests remain in the queue for the next epoch.

Rewarding mechanism: we apply the following rewarding scheme to reflect our favor for immediate service and sleeping in low-traffic periods. For each "waiting" request, E emits R_s as service reward if this request is served, R_q if queued or retransmitted, and R_f if it fails. In each epoch, S emits C_1 as the operational cost if it is turned on and C_0 if it sleeps.

4. EVALUATION RESULTS

We evaluate the operation of our implementation using a BS sleeping algorithm based on deep Q learning [10]. The goal is to show that the algorithm can learn a reasonable sleeping policy with the traffic generated by our emulator and that the internal state of our emulator can by dynamically modified in reaction to different control commands. The parameters used are: $R_s = 1$, $R_q = -1$, $R_f = -10$, $p = 0.7$, $C_1 = 1$, $C_0 = 0$. Fig. 5 shows the number of sessions and requests issued during the test. As can be seen, the peaks and valleys in the two time series match with each other, which is the expected result because we want the generated traffic to be mainly modulated by the captured dataset.

To illustrate the learning process of the controlling agent, we show the percentage of waken epochs as well as the number of generated requests in each 5-minute period in Fig. 6. As can be seen, the agent's control actions does not make much sense at the beginning. At the beginning, it wakes up and sleeps equally randomly. A while later, it grabs an apparently sub-optimal policy by which it stays up regard less of the traffic. The agent then spends the first 50 minute improving its policy. After the first hour, the agent gradually grasps a reasonably good policy and start to make sensible moves, i.e. sleep with high probability when the traffic is low and turn on itself when the traffic becomes heavy.

To illustrate users' reactions to BS sleeping, we show in Fig. 7 the total number of waiting and failed requests in each 5-minute period. As can be seen, the number of waiting requests is high in low-traffic periods and low in high-

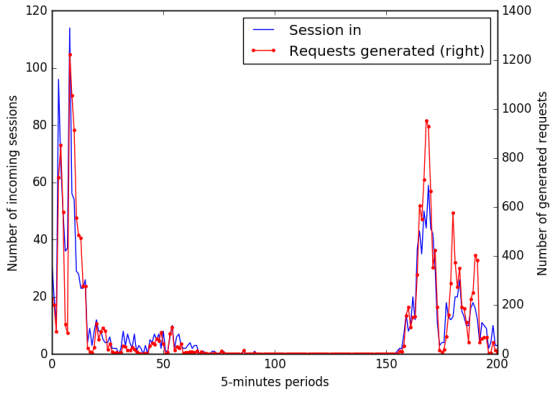

Figure 5: Number of sessions and requests generated during a 1000-minute period.

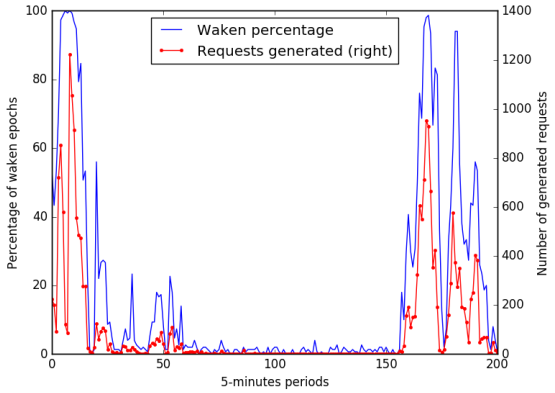

Figure 6: Percentage of waken epochs and number of requests generated during a 1000-minute period.

traffic periods. This is expected as the BS will sleep more in low-traffic periods and more users will choose to wait in response, while the BS is likely to stay on in peak hour to immediately serve all incoming traffic. Also, notice a small portion of request will time out and fail in low-traffic periods due to long waiting time.

5. RELATED WORK

The notion "network testing" actually entails two complementary aspects: the emulation of the network functions and the generation of traffic. Each of the two have a few different methods with different level of trade-off among fidelity, reproducibility, and cost-effectiveness. The choice of combination is often based on the primary goal and available resource in each testing task. In this section we briefly review related work in both aspects and compare with our proposal.

The simulation or emulation of network function is the first pillar of network testing. Simulation and emulation is often combined to form a meta emulation system: emulation is used to abstract the interface of the network functions, while simulation is used to simplify the process by generating

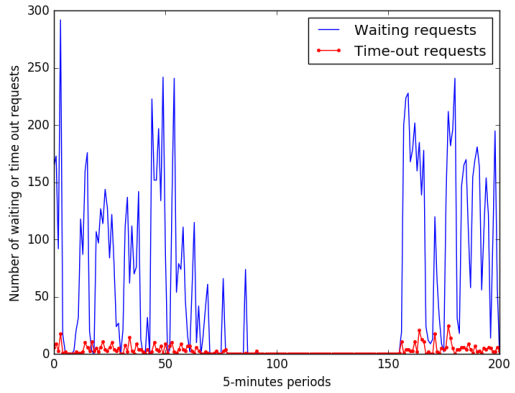

Figure 7: Number of waiting and failed requests during a 1000-minute period.

the aggregate behavior of network functions using models. An example is the evaluation of rate-adaption algorithms in wireless networks. The phenomenons related to wireless propagation and physical-layer network functions are implemented using analytical models. Their collective behavior is then wrapped up following the function interface specifications and interact with the algorithm under test. Our approach for network emulation is following the same general framework, despite that the integration of reinforcement learning requires us to separately implement a control plane (D) and a data plane (S). We will not dive deep into this topic. For interested readers, a thorough review of network simulation and emulation is presented in [9].

Existing traffic generators can be roughly classified as either model-based or trace-based [11]. Model-based generators generate traffic using predefined analytical models such as periodical and Poisson. There exist a number of well-developed software packages, e.g. TG[4] and MGEN[5]. Many widely used network emulation platforms also have embedded module for this purpose. However, it is often hard to develop an analytical or computationally tractable model that closely fit to real traffic sources. Trace-based generators, on the other hand, solves this problem by modulating the traffic generation process with realistic traffic traces collected from real networks. This is achieved by either replaying captured traffic traces or first distilling traffic model parameters from the collected trace and then synthesizing a new trace following the parameterized model.

The biggest problem with trace-driven methods is the reaction gap as described previously. Different manifestations of this problem has been independently identified and attacked in various fields. For Internet traffic generation, Hong et al. [5] first proposed in TCPopera to emulate a full TCP protocol stack when generating traffic. In this way, trace packets that breaks TCP sementics (such as "ghost" packets) can be filtered out. Also, if the packet drop rate or delay of the test-

[4]By ISI of USC. `http://www.postel.org/tg/`
[5]By Naval Research Laboratory (NRL) PROTocol Engineering Advanced Networking (PROTEAN) group. `http://www.nrl.navy.mil/itd/ncs/products/mgen`

ing network is different from that of collection, the emulated retransmission and congestion control functions will dynamically adjust the transmitting schedule to interactively react to the changing environment. For wireless networsk, a state-machine-based method is proposed in [7] to coordinate the generated WLAN traffic with environmental effects such as fading and noise. For System-on-Chip (SoC) networking, the problem is centered around how to generate traffic at the right pace if the target emulation system has different clock frequency [4]. For application-layer Internet traffic, the goal is to identify and adjust session-depended information fields such as host-name and cookie. Two different approaches are proposed in [2] and [12] to modify those fields according to the network context.

The problems in generating WLAN, SoC, and application-layer traffic are however static in nature: no dynamic control for the replaying process is required. In this sense, the probelm we try to solve is more related to the one dealt in [5, 1]. Still, one major difference lies in that we try to model the response of users while the two pieces of work emulates the response of the protocol machinery. This piece of difference also makes our work complimentary to previous work, because by jointly modeling the response of both human and machinery, the traffic generation process can achieve even higher realism. Besides, there are also two minor differences: our proposal emits rewards in real-time in response to network control decisions, which is a required for testing reinforcement learning algorithms; also, none of these previous works uses application-level traces collected from a wireless network.

6. CONCLUSIONS

In this paper, we propose a data-driven interactive network emulation framework called DragonEye. It is designed to enable interactive data-driven tests for network control algorithms, especially the ones that fit into the reinforcement learning formulation, for cellular networks. We discuss the problems with naive trace replaying and summarize them as the refinement and reaction gaps. DragonEye mitigates these two gaps by combining data-modulated emulation with microscopic refining and reacting models. We describe the components and operation of DragonEye and implement it using real dataset collected from a campus WLAN. Then we evaluate this implementation by testing a deep-Q-learning-based BS sleeping agent. Results show that the emulation process is indeed reactive to the dynamic sleeping operations of the serving BS, which demonstrates the effectiveness of DragonEye in emulating live interaction using offline data.

7. ACKNOWLEDGEMENTS

This work is sponsored in part by the National Basic Research Program of China (973 Program: 2012CB316001), the National Science Foundation of China (NSFC) under grant No. 61201191, No. 61321061, No. 61401250, and No. 61461136004, the Creative Research Groups of NSFC (No. 61321061), the Sino-Finnish Joint Research Program of NSFC (No. 61461136004), and Hitachi RD Headquarter. We also want to express our sincere appreciation for Shanghai Jiaotong University to provide all the datasets used in this work.

8. REFERENCES

[1] W. Chu, X. Guan, Z. Cai, and L. Gao. Real-time volume control for interactive network traffic replay. *Comput. Netw.*, 57(7):1611–1629, 2013.

[2] W. Cui, V. Paxson, N. C. Weaver, and Y. H. Katz. Protocol-independent adaptive replay of application dialog. In *Network and Distributed System Security Symposium 2006*, 2006.

[3] M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi. Understanding individual human mobility patterns. *Nature*, 453(7196):779–782, 2008.

[4] J. Hestness, B. Grot, and S. W. Keckler. Netrace: dependency-driven trace-based network-on-chip simulation. In *Proceedings of the Third International Workshop on Network on Chip Architectures*, 2010.

[5] S.-S. Hong and S. F. Wu. On interactive internet traffic replay. In *RAID*, volume 3858, pages 247–264. Springer, 2005.

[6] U. Javed, D. Han, R. Caceres, J. Pang, S. Seshan, and A. Varshavsky. Predicting handoffs in 3G networks. *SIGOPS Oper. Syst. Rev.*, 45(3):65–70, 2012.

[7] C.-Y. Ku, Y.-D. Lin, Y.-C. Lai, P.-H. Li, and K.-J. Lin. Real traffic replay over WLAN with environment emulation. In *2012 IEEE Wireless Communications and Networking Conference (WCNC)*, 2012.

[8] J. Liu, R. Deng, S. Zhou, and Z. Niu. Seeing the unobservable: channel learning for wireless communication networks. In *Proceedings of the 2015 Global Communications Conference (Globecom)*, 2015.

[9] E. Lochin, T. PÃl'rennou, and L. Dairaine. When should I use network emulation? *Annals of Telecommunications*, 67(5):247–255, 2011.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[11] S. Molnar, P. Megyesi, and G. Szabo. How to validate traffic generators? In *2013 IEEE International Conference on Communications Workshops (ICC)*, 2013.

[12] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: automatic protocol replay by binary analysis. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[13] E. Oh, B. Krishnamachari, X. Liu, and Z. Niu. Toward dynamic energy-efficient operation of cellular network infrastructure. *IEEE Communications Magazine*, 49(6):56–61, 2011.

[14] R. S. Sutton and A. G. Barto. *Introduction to reinforcement learning*, volume 135. MIT Press Cambridge, 1998.

[15] F. P. Tso, J. Teng, W. Jia, and D. Xuan. Mobility: a double-edged sword for HSPA networks: a large-scale test on hong kong mobile HSPA networks. *IEEE Transactions on Parallel and Distributed Systems*, 23(10):1895–1907, 2012.

[16] J. Wu, S. Zhou, and Z. Niu. Traffic-aware base station sleeping control and power matching for energy-delay tradeoffs in green cellular networks. *IEEE Transactions on Wireless Communications*, 12(8):4196–4209, August 2013.