A Comprehensive Framework to Evaluate Wireless Networks in Simulation and Real Systems

Martin Backhaus and Markus Theil and Michael Rossberg
and Guenter Schaefer and David Sukiennik

Telematics and Computer Networks Research Group
Technische Universität Ilmenau, Germany
[martin.backhaus, markus.theil, michael.rossberg, guenter.schaefer, david.sukiennik][at]tu-ilmenau.de

***Abstract*—A thorough performance evaluation of protocols and algorithms for (wireless) networks requires simulation and real-system experiments, as both of them provide individual benefits. Usually, this calls for two separate implementations: One tailored to a discrete-event simulator and a second designed to run on real hardware. Therefore, significant effort is required to implement the same mechanisms or protocols twice. To avoid this overhead, we propose a comprehensive framework based on DPDK and OMNeT++, allowing to run simulations and real-system experiments from the very same codebase. Hence the best of both worlds is available: scalable scenarios and reproducibility when simulating, and realistic behavior and real-world performance metrics when running real-system experiments. Our evaluation of several representative real-world networking scenarios analyzes similarities between simulation and real-system results and discusses the framework qualitatively. Quantitative results indicate that the approach performs well, i.e., it allows even for productive deployment using the codebase later on, and results from both worlds are comparable.**

***Index Terms*—Wireless Network Evaluation Framework, Simulation, Real System, OMNeT++, DPDK**

I. Introduction

When the first IEEE 802.11 standard was released over 20 years ago, it consisted of rather simple algorithms and protocol mechanisms. However, as of today, the Wi-Fi standard has not only become much more complex itself, but novel use cases arose, e.g., Internet of Things (IoT) and applications in industrial automation, police operations, disaster scenarios, or large-area mesh networks (see Freifunk [1]). With an associated increase in complexity, the development and debugging of protocols (especially for large-scale networks) becomes a rather challenging task – particularly when examining throughput for multiple demands, robust routing, Quality of Service (QoS) guarantees, security aspects, etc. As for now, drivers and IEEE 802.11 [2] stacks are usually implemented by kernel components of operating systems, therefore, leading to difficult debugging characteristics and complexity issues. General-purpose operating systems additionally imply performance drawbacks in some use cases, as they are not primarily tuned for packet forwarding.

This work was supported by RWTÜV-Stiftung (Essen, Germany) under grant number S189/10029/2016.

Concerning research targeted at Wireless Mesh Networks (WMNs), current articles accompanied by experiments are usually either based on idealized simulation studies or non-reproducible results in small, real-world testbeds. Recently, Papadopoulos et al. [3] performed a literature study to obtain data on how current research articles evaluate the performance of Wireless Sensor Networks (WSNs). They found out that only 20.3% of relevant articles contain both simulation results and an experimental setup using real hardware. When Uludag et al. [4] compared existing WMN testbeds, they also stated integration with simulators, monitoring, debugging techniques and pre-deployment testing as open issues. In general, both simulation and experiment are indispensable for a thorough evaluation of protocols in WMNs. On one hand, simulation not only enables reproducibility of test cases, but more importantly facilitates experiments with different topologies and large networks with relatively low effort. On the other hand, experimenting with real hardware is a necessity, too, as it allows to verify the protocols (or applications) behavior under real network conditions and especially the influence of parallel execution (if desired). In particular, only performance benchmarks conducted on real hardware may provide a solid indication whether or not a system can be employed in specific use cases. And last, environmental conditions (e.g. interference, shadowing) cannot easily be simulated in sufficient detail to evaluate wireless networks realistically.

Consequently, a thorough development and evaluation of a wireless protocol (or any other approach) demands both simulation and real-system experiments – which usually requires

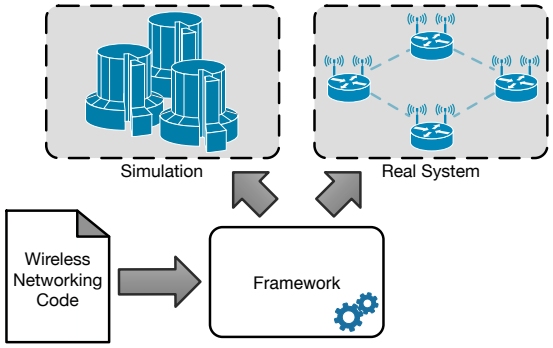


Fig. 1. Key concept of a framework for combined simulation and real-world deployment.

two separate implementations, resulting in significant effort due to substantial differences between the coding paradigms. This is why this paper proposes a framework that enables experimenting in both – a simulative environment and real-world operation – and that is using the very same codebase for both cases. Fig. 1 illustrates the key concept of the evolving framework. It depicts a middleware, adapting programming paradigms of specific real-system code to simulation code. Our framework has similarities to model-driven design principles: restrictions to the user-defined code are being introduced to comply with the framework's function. In contrast, restrictions imposed by our framework are rather insignificant and we provide full control and debugging capabilities, which we will prove over the course of this paper.

In particular, we provide the following contributions:

1) a comprehensive framework for experimenting with wireless applications and protocols in simulation and real system using the same codebase and
2) an evaluation of the framework to assess its general performance and to point out similarities (or differences) when executing code in the aforementioned twofold way.

The framework (and potential applications utilizing it) significantly benefit from employing Network Function Virtualization (NFV) techniques, as they enable user-space development on the one hand and high-performance frame processing on the other hand. This allows rapidly implementing even complex functionalities regarding Wireless Local Area Network (WLAN) frame processing for specialized needs and achieving high data rates as well as low delays. Therefore, we rely on our previous work in [5], where we integrated a wireless network interface into the Data Plane Development Kit (DPDK) – a fast packet I/O framework.

The remainder of this paper is organized as follows: First, we outline an overview of state-of-the-art approaches on unifying simulation and real system and we motivate requirements for our resulting approach (Sec. II). We present our framework in Sec. III in detail and reveal its inner workings. The evaluation in Sec. IV discusses our approach qualitatively and studies its performance as well as applicability, and we check simulation and real-system results for their significance using small benchmark scenarios. Summing up, Sec. V presents some concluding thoughts and directions for future research.

II. Requirements and Related Work

For the approach to work, it needs to fulfill the following functional requirements:

1) The framework's key requirement is the twofold aim of running applications in simulation and real experiment.
2) It should provide access to wired and wireless network interfaces, i.e., handling of IEEE 802.3 & 802.11 Medium Access Control (MAC) protocols and
3) access to MAC functions of wireless Network Interface Controllers (NICs).
4) Control key aspects of physical-layer operation (channel, number of streams, coding options) shall be considered.
5) Possibility to run different code on different nodes must be given.

Besides, the subsequently listed non-functional requirements need to be considered as well:

1) Real system's performance should not degrade when using the framework (compared to plain DPDK applications). That means, performance of the real system is paramount when joining both paradigms – drawbacks for the simulation's runtime behavior are acceptable. Although simulation processes would benefit from performance optimizations as well, design decisions should constantly favor real system's performance, as the performance of an approach or a protocol that is being evaluated with the framework will not be influenced by a slow simulation environment.
2) Minimal restrictions to user-defined code for approaches to be evaluated with the framework
3) Comparable results in simulation and real system
4) Extensibility & easy integration of new hardware
5) "Comfortable" development and debugging

When looking at state-of-the-art approaches, there is some prior work with similar objectives. Mayer et al. [6] discussed the concept of integrating existing software into the OMNeT++ simulator by shared-library calls. When compiling the program, they conditionally use functions related to time: either those of simulator or those from the operating system. Networking applications have to deal with the operating system's networking-related system calls or those of OMNeT++. In general, the resulting code has to be aware of the simulator's programming paradigm, which implies significant changes to existing networking code or strong restrictions for newly developed implementations. Naumann et al. [7] propose a similar method for integration: They suggest intercepting network and timing-related system calls and connecting them to the simulator.

However, both aforementioned state-of-the-art approaches struggle with parallelism or the integration of networking code in kernel mode. Therefore, it is unclear if a dualism between simulation and real system is adequate concerning comparability of results from both worlds and the quantity of restrictions to the code.

Another concept [8] uses a testbed consisting of Virtual Machines (VMs), executing applications and kernel code. The interconnection of VMs is handled by a simulation node running OMNeT++. VMs run a virtual wireless NIC kernel module, which sends all frames to the simulator. The simulator then introduces delay and packet loss and sends the frames back to the target node, which injects them in the kernel via the virtual wireless NIC again. It is unclear how this concept performs in larger scenarios, as the authors only used five virtual nodes in their evaluation. The single-threaded design of a discrete-event simulator becomes a bottleneck and functional barrier. Although a discrete-event simulator is used, the setup will not lead to reproducible experiments.

All aforementioned approaches aim at integrating existing real-system code into simulators or combined experiments, which is not as comprehensive as our objective of the twofold way of executing experiments from an identical codebase.

MathWorks offers the option for discrete-event simulation using Simulink [9] and analysis of the physical layer in WLANs [10] using MATLAB. While the simulation of the physical layer is – in terms of detail and accuracy – superior to common network simulators, Simulink does not include common protocols and functionality of higher network layers (e.g. MAC, routing) that are needed for a thorough evaluation of IEEE 802.11 networks. Furthermore, the programming language, Application Programming Interfaces (APIs) and paradigms differ strongly from established concepts in network-application development, impeding the development of network applications.

To the best of our knowledge, the approach presented in this paper is the first one to lay a foundation for experimenting with wireless protocols and approaches, which enables evaluation in both reproducible simulation and high-performance real system with the exact same codebase.

III. The Framework

This section introduces our framework in detail, including fundamental design aspects as well as its inner workings. We start by describing problems to be encountered when joining simulation and real system and present suitable solutions. Specifics to both worlds and their relevance are described as well.

A. Design Decisions and Architecture

As real system's performance is one of the main requirements for the evolved framework, we decided to take our previous work in [5] as a foundation. It introduces a flexible user-space architecture for IEEE 802.11-frame processing using DPDK and offers two main advantages:

1) High-performance processing of WLAN frames.
2) Easy debugging and experimenting due to user-space implementations.

Gathering those benefits requires the user to write DPDK-like applications. Consequently, our framework needs to provide adequate adaptations from DPDK code to the simulation environment and should be implemented in C/C++. For performance reasons and due to the complexity of integrating two paradigms, we chose C++. In particular, we transformed relevant parts of the DPDK-API: packet representation and manipulation, reception and transmission of packets, and timers. These functionalities build a reasonable foundation for any networking application, but further components of the API can be adapted whenever necessary. Additionally, we included adaptations for channel configuration that were added as component of IEEE 802.11 management during prior work. These adaptations allow channel configuration to work in a similar fashion in both simulation and real system.

When joining fundamentally different paradigms of simulators and real-world networking applications, some issues arise compared to the case of using code tailored to either simulation or real system:

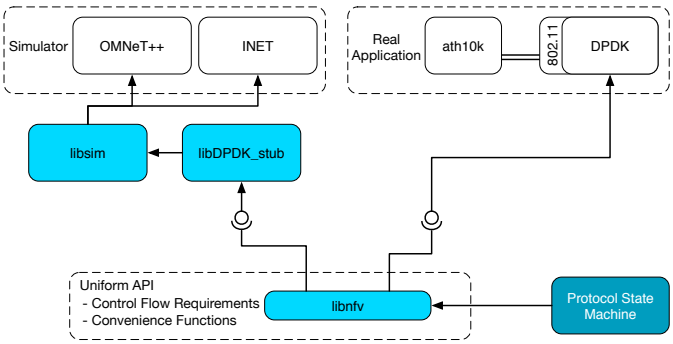


Fig. 2. Overview of framework architecture. In white existing components, light blue newly designed components, and user-defined applications/components in dark blue.

- *Control Flow:* Simulation programs have a fundamental difference from DPDK applications concerning control flow, which the framework needs to overcome. Packet frames are handled in an endless loop in DPDK on one node, while discrete-event simulators handle events for many different nodes in the same process.
- *Number of Applications:* In a real system, one compute node runs one executable; whereas in a simulation, many nodes with the same code reside in the executable representing the simulation itself. This problem is aggravated due to the fact that real-system code is usually unaware of simulation aspects. Additionally, conventional executables cannot be integrated in simulation environments.
- *Different Application Types:* Similar to the previous issue, separation of different applications (e.g. client/server software, protocol state machines) is achieved by using different compute nodes. A simulation environment needs dedicated mechanisms to incorporate arbitrarily many types of different applications that are unaware of simulator APIs.
- *Parallel Execution:* When looking at DPDK applications, the degree of parallelism strongly influences achievable performance. In contrast to this, most simulation code in discrete-event simulators follows a strictly sequential pattern. This may not be relevant to performance evaluation in a simulation, as calculations in an application do not require simulated time. But for suitable adaptations in the context of our framework, it is important to support DPDK's multi-threading capabilities, in order not to reduce the user's possibilities to write high-performance applications.

The framework's architecture based on different components accounting for simulation adaptations, real application elements, and shared functionalities is depicted in Fig. 2. Both simulation and real-system characteristics are mapped to the `libnfv` component, which demands a certain control flow and provides convenience functions. `libDPDK_stub` represents relevant parts of DPDK's API that need to be transformed to work with

a simulator and `libsim` takes care of control-flow adaptations and parallelism.

Two of the most popular and up-to-date discrete-event network simulators are ns-3 [11] and OMNeT++, combined with the model library INET [12]. Both offer similar abstractions and concepts. At the time of decision-making for this work, ns-3 and OMNeT++/INET supported wired networks alike and differed slightly in terms of IEEE 802.11 support. Either simulator required contributions to allow for simulation of modern WLANs, i.e., up to IEEE 802.11ac. We chose OMNeT++ and INET for the underlying simulator – as depicted in the top left corner of Fig. 2. On the top right, we indicate the use of our aforementioned work in [5], which extends DPDK by relevant IEEE 802.11-management functionalities and support for wireless interfaces of the `ath10k` family. User-defined code, e.g., a protocol state machine as seen on the bottom right, uses `libnfv`-abstractions only.

B. Control Flow

Concerning control flow, the use of endless loops must be prohibited when simulating many nodes running DPDK-like code. Otherwise, a starting node will not suspend its computations after initialization, unless interrupted by complex mechanisms. For simplicity, we decided to demand the user to provide the body of the endless loop in a dedicated function: `iterate()`. Therefore, DPDK code can be executed iteration by iteration, one node at a time. The number of iterations needs to be chosen appropriately, leading to the following prerequisite: further iterations are not necessary, if the last iteration did not receive or send packets. Consequently, iterations will be scheduled based on packet reception and transmission, as well as timer events. To complete the control flow of the applications lifetime, functions for initialization and finalization will be added as well: `init()` and `finish()`. Concerning DPDK applications, this restriction is not significant, as the majority of DPDK code almost follows this particular programming pattern (aside from functions instead of class methods and different naming conventions). The only notable modification is the transformation of the endless loop into a single function for explicit iteration of the former loop body. This control flow results in the following interface, which picks up naming conventions from DPDK, where the abstraction of threads is called logical core (lcore):

```
class LCoreFunctionBase {
  public:
    virtual void run() = 0;
    virtual void init() = 0;
    virtual void finish() = 0;
#ifdef DPDK_CODE
    void iterate();
#else
    virtual void iterate() = 0;
#endif
};
```

Please note that in case of use with DPDK, `iterate()` is non-virtual for performance reasons. Detailed explanations will follow in the remainder of this section.

C. Simulation

Simulators enable analyzing the behavior of large-scale networks and different topologies with relatively low effort. As mentioned before, the simulation environment of choice (OMNeT++) needs to be extended to work with code using a DPDK-like API for network interfaces and control-related parts (e.g. timers and multi-threading functionalities).

At first glance, different concepts of time (continuous vs. discrete) are not compatible with each other and demand further measures to be taken. After the already outlined transformation of an endless loop into a callable function relating to the former loop body, one iteration can be associated to simulator events. These events simply need adequate scheduling to create the desired behavior of the application, which we already discussed in Subsection III-B. In case an application creates a disproportional quantity of packets (e.g. a packet generator transmitting in every single iteration), a "queue-full" status serves our framework as justification to preempt the applications execution, i.e., no further iterations will be scheduled.

The uniform, previously described API helps tackling the issues concerning the number and variety of applications. Many applications of the same type can be realized by several instances of the same class representing the application. This implies prohibition of global variables – a clean separation of different instances is at risk otherwise. Different applications are handled the same way, but for modularity reasons, we decided to build applications as shared libraries. These are being loaded dynamically into the simulation and since they all follow the same API, they can be easily integrated.

Concerning parallelism, we made sure that our lcore-abstraction and associated functions behave similar to the ones in DPDK, i.e., dynamic creation and execution of many lcores is possible. Due to the single-threaded design of discrete-event simulators, every execution intended in parallel is serialized into one thread. This implies 1) debugging capabilities of the simulation with respect to multi-threading are limited and 2) the behavior of multi-threaded applications can inherently not be reproduced. Although this is a drawback of discrete-event simulation, it leads to a significant advantage of our framework, since testing in both worlds is easily possible.

Simulation Awareness: Due to the assumption concerning the number of necessary iterations for each application, some real-system programming practices do not work straightaway when simulating. One noteworthy example is "busy waiting" until a certain time has passed. In a real-system application, this is relatively easy and does not involve potentially complex timer implementations (that may degrade performance). During simulation and due to the mentioned assumptions, busy waiting does not work as expected: after one iteration, no packets are generated – leading to no further iterations and hereby stopping the process of busy waiting. This is why we introduced functions for simulation awareness designed to solve problems from real-system programming patterns such as busy waiting. In this case, a function `setDelayedWork(double delay)` leads to another iteration after `delay` seconds have passed.

Other events that may occur in between, e.g., reception of packets and timer events, lead to immediate iterations without waiting for the desired time to pass.

We present an outline of simulation awareness functions:

- `setDelayedWork(double delay)`: As described above.
- `setToMoreImmediateWork()`: In case one more iteration is desired immediately.
- `setFinished()`: For application termination.

Please note that all scenarios where simulation awareness was applied could have been solved using timers. Therefore, the newly introduced functions for simulation awareness merely depict a way of code optimization. In case of a real-system build, simulation awareness functions are defined as empty functions that will be optimized out by the compiler.

Another option to solve the problem completely, is to execute iterations on a regular basis including a small, randomized processing time that advances simulation time as well. But that would lead to unacceptable overhead during simulation – especially in large networks possibly involving numerous applications.

A domain where simulation awareness might gain importance are hardware signals, which are not considered in the current framework implementation. They will be subject to future work.

D. Real System

When writing an application using the framework, users have to subclass `LCoreFunction`, which itself subclasses `LCoreFunctionBase`, or in simplified form:

```
template <class T>
class LCoreFunction : public LCoreFunctionBase {
  int run() override {
    init();
    while(not hasFinished()) {
      static_cast<T*>(this)->iterate();
    }
    finish();
    return returnCode();
  }
};
```

The class `LCoreFunction` is implemented using the "Curiously Recurring Template Pattern (CRTP)" [13], which facilitates compile-time polymorphism, i.e., it avoids the overhead of virtual function calls. The former control flow (including the endless loop of the real system's application) translates to `run()`, which calls the user-implemented `init()` and then repeats iterations. At last it calls the user's `finish()`.

A possible user's implementation could be designed like:

```
class Worker : public LCoreFunction<Worker> {
  public:
  void iterate() {
      handleTimers();
      rx();
      tx();
    }
};
```

It depicts a high-level example of one iteration: timers will be handled and packets received, processed and transmitted (if needed). Due to the definition of `iterate()` inside the class body, it is implicitly an inline function, allowing avoidance of function call overhead. Leveraged by the CRTP, `iterate()` can be resolved during compilation and benefit from further optimizations by the compiler and linker, diminishing performance penalties that may arise from our abstraction layer otherwise.

E. Summary of Restrictions and Assumptions

For clarity, we would like to summarize all relevant restrictions to the user-defined code briefly:

- Uniform control flow consisting of initialization, iteration and finalization of the application (see Subsection III-B).
- No endless loops allowed.
- No global variables allowed.

Furthermore, it must be known how often these iterations need to be done, as the original behavior (endless iteration) is not an option for simulation of network nodes. The suitable criterion for termination of iterations relates to packet reception and transmission, as well as timer events. If there is an iteration with none of the aforementioned incidences, we presume no immediate need for further iterations. Subsequent simulation events can of course lead to more iterations later on.

IV. Evaluation

This section presents the test setup and scenarios in detail, as well as the metrics used in our quantitative evaluation. Any given setting is evaluated using 32 repetitions for statistically significant results. Functional and non-functional requirements are appropriately covered in a qualitative or quantitative evaluation respectively.

A. Qualitative Discussion

Following the list of requirements stated in Sec. II our framework needs to provide access to NICs on low level (Ethernet frames and 802.11 frames). This is possible due to the integration of DPDK and previous work. The simulation integration with OMNeT++/INET also provides the possibility to access frames on MAC layer. Some restrictions arise on real hardware, due to the closed-source nature of WLAN firmware.

Simulation as well as DPDK allow the configuration of the most important physical-layer settings, like channel, number of spatial streams, coding, and aggregation options.

The framework and accompanying tools enable deployment of different network functions to our testbed as static binaries or execution in the simulation, with a shared library per node. If global variables are omitted and our API is used, the code of each node can run independently, without further restrictions on the code organization. As desired, user-defined code following our API can be deployed for both simulative evaluation and experiments in real systems (as we did for the quantitative evaluation process later on).

Moving on to non-functional requirements, the real system's performance has to be similar compared to plain DPDK applications. The usage of the CRTP [13] omits performance penalties induced by virtual function calls through employing

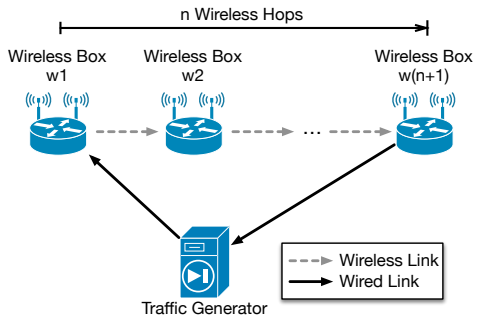


Fig. 3. Experimental setup for evaluation: scenario 1 (multi hop).

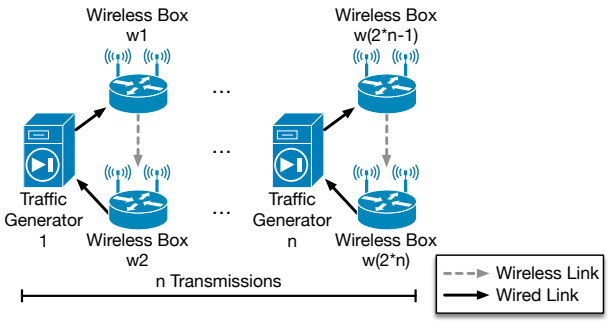


Fig. 4. Experimental setup for evaluation: scenario 2 (multiple transmissions).

compile-time polymorphism. Therefore, we do not expect any performance drawbacks compared to usual DPDK applications when employing the framework. Small benchmarks (not included in this paper for brevity) confirmed our expectation.

DPDK is already acknowledged in the research community as a scalable framework for network processing. In our previous work, we have shown that IEEE 802.11 frames can also be processed efficiently with DPDK in user space. The restrictions our framework imposes are minimal and do not impede developers/researchers much compared to plain DPDK programming.

DPDK and OMNeT++/INET provide abstractions for hardware and network protocols and can be easily extended. Integration of new network hardware is also possible, as our framework is only a small abstraction layer and allows the use of new NIC drivers as soon as they are added to DPDK.

Debugging of network protocol implementations is simplified, as user-space debugging tools can be used to examine performance with DPDK and large scenarios with OMNeT++. The simulation also provides reproducible experiments, which can also be critical for debugging purposes.

B. Experimental Setup, Scenarios and Metrics

Our setup covers two representative networking scenarios:

Scenario 1 – Multi Hop: The first scenario consists of two to four WLAN nodes in a row, as depicted in Fig. 3, i.e., the number of wireless hops n will range between 1 and 3. A traffic generator is connected to the first and the last node and sends time-stamped UDP packets through the network. All WLAN nodes run a rather simple forwarding application with appropriately configured forwarding tables. Furthermore, all wireless nodes are in communication range of each other due to restricted space during our real-world tests. Due to appropriate configuration of forwarding tables, nodes only forward data to the next or previous node in the row.

Scenario 2 – Multiple Transmissions: Fig. 4 displays the setup for multiple transmissions happening simultaneously. It consists of n pairs of wireless nodes, each pair connected with a separate traffic generator. Consequently, the scenario resembles n replications of scenario 1 in case of a single wireless hop. All wireless nodes are within each other's communication reach. Applications are used as described in the previous scenario and all forwarding tables configured to allow for an according packet flow as the figure depicts. We employed suitable time-synchronization methods for all traffic generators to make sure that there were no problems concerning experiment start and duration.

In general, both traffic generator and wireless forwarding applications are implemented using our evaluation framework in all scenarios. Therefore, the exact same configuration files can be used in simulation and on real hardware within one scenario. In the simulation, a corresponding topology is created for OMNeT++/INET and the code of each node is encapsulated in a shared library. Traffic is sent unidirectionally in all experiments. With source and destination interfaces of the generated traffic residing on the same machine, one-way delay measurements and evaluation of test cases are particularly easy to perform. One-way delay and packets per second will be figures of merit for our quantitative evaluation. The traffic generator counts successfully transmitted packets during the measurement and records their one-way delay values. For the second scenario (examining multiple transmissions), the process is the same for each individual traffic generator. Finalizing the experiment requires an adequate aggregation of metric values among generators in the following fashion: for transmitted packets, we calculate the sum of the generator averages (resembling a kind of network throughput) and concerning delay, averages will be averaged again.

Every component has a separate wired management interface, which is managed by the operating system to allow SSH connections – since this is normal for an experimental testbed and we do not use the interface otherwise, we will omit it in any further discussion. Each wireless box is equipped with a Wi-Fi module based on the QCA9888 chipset (Compex WLE650V5-18 IEEE 802.11ac Wave 2 adapter with two antennas). Considering our needs for a wireless testbed, we chose PC Engines APU2C4 embedded boards, which feature multiple 1 GbE interfaces (based on Intel i210 chipset) and a x86_64 AMD CPU with 4 cores at 1.0 GHz. The traffic generator is also based on an APU2C4 board. DPDK supports i210 based NICs and the CPU was tested to be capable of generating 1 GBit/s with wire speed in any given case.

Table I summarizes relevant settings and software versions in our experiments. Settings subject to change over the course of our evaluation (factors) are marked with "(F)". If simulation and real system diverge in their behavior, this would become most evident under high traffic load. Therefore, we apply a

TABLE I
MEASUREMENT SETTINGS AND EXPERIMENT FACTORS (F)

Setting/Factor	Used Values
Measurement Runs	32 per factor combination
Measurement Duration	30 s
Scenario (F)	1) Multi-hop communication 2) Multiple transmissions
No. of Wireless Hops (F)	1, 2, 3; relevant for scenario 1
No. of Transmissions (F)	1, 2, 3; relevant for scenario 2
Mode	IEEE 802.11a
Encryption	None
Operating Mode	Mesh Point
Beacon Interval	1000 ms
Transport Protocol	UDP
Datagram Size	60 Byte
Datagram Rate	10 000 $1/s$
Linux Distribution	Debian GNU/Linux 9 (stretch)
Linux Kernel	4.16.5
DPDK	17.11
OMNeT++	5.2
INET Framework	3.6.2
Measurement Tool	DPDK-based traffic generator, same code for OMNeT++ and testbed

higher load than the system could theoretically cope with: utilization of small UDP packets at a high rate.

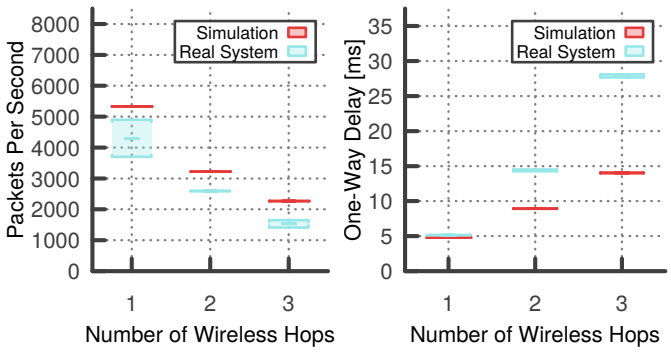
Both INET and the real hardware were configured to operate in 802.11a mode. It is clearly not optimal to use an older operation mode, but it suffices to discuss the key aspect of our framework: comparability of results between simulation and real systems. We started implementing 802.11ac in INET to compare it to our 802.11ac capable NICs and, furthermore, to investigate if the framework can cope with high throughput. INET 3.6.2 does not support vital mechanisms relevant for 802.11ac's performance, e.g., MPDU aggregation, advanced coding schemes, or multiple spatial streams. Hence, we were not able to achieve similar performance levels with INET, as with our QCA9888 based NICs in case of 802.11ac.

Our driver currently does not utilize intermediate queues per wireless peer, as the Linux kernel does. The direct use of a single large queue of outstanding transmission packets with over 2000 packet buffers leads to unusually high delays ($\gg$ 100 ms) when using comparatively low data rates of IEEE 802.11a. A common approach to deal with high delays due to bufferbloat [14] is queue-length bounding, which [15] discusses for wireless networks. In order to obtain comparable results, we adjusted the queue length of the `ath10k` driver.

C. Quantitative Results

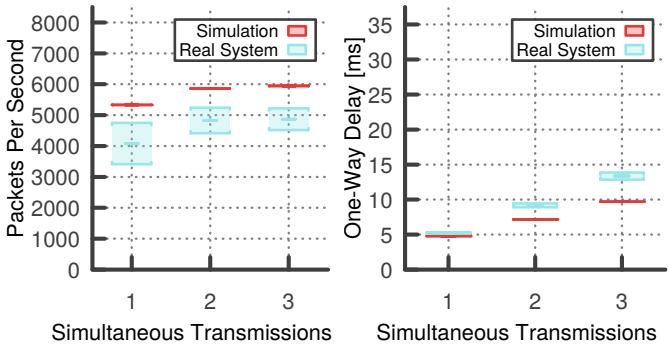
The following quantitative evaluation mainly covers comparability of results when using our framework in both simulation and real system in a representative scenario introduced in the previous subsection. A general conjecture is that since simulations depict somewhat idealized scenarios, performance metrics of the real system are supposed to be slightly worse.

Scenario 1: Fig. 5(a) depicts the packet rate in relation to the hop number. The number of transmitted packets per second is consistently higher in the simulation, but of comparable magnitude. Results from our testbed have higher variation



(a) Packet rate. (b) Delay.

Fig. 5. Achieved metrics for scenario 1 – multi-hop communication (60 Byte packets, 99% confidence intervals).



(a) Packet rate. (b) Delay.

Fig. 6. Achieved metrics for scenario 2 – multiple transmissions (60 Byte packets, 99% confidence intervals).

because of environmental effects. Let n be the number of wireless hops. As all nodes are in communication reach of each other, the packet rate of the direct case decreases approximately with $1/n$ for two and three hops. Resulting delays depending on the number of wireless hops can be obtained from Fig. 5(b). Delay in the simulation increases with growing number of wireless hops n. On real hardware, it is higher in any given case. As already mentioned in Subsection IV-B, we have adjusted the software queue length between both environments in the direct case ($n = 1$). We suppose that delays for $n > 1$ can be brought closer together by adjusting the behavior of OMNeT++ and INET, which demands further analysis in the future. Each additional hop in the real system adds roughly 5 ms to the metric value obtained from simulation value (e.g. 15 ms in case of $n = 3$ hops). Most likely, this divergence results from some form of queuing in the firmware/hardware of the cards, which we cannot control.

Scenario 2: Fig. 6(a) illustrates the sum of transmitted packets from all traffic generators. Values are shown for one to three simultaneous transmissions. Most important for applicability of our framework are values of similar magnitude for simulation and real system. As in the previous scenario, the simulation is significantly better due to its idealized character. In order to understand the increase in the overall quantity of packets per second with an increasing number of simultaneous transmissions, we need to recall the behavior of

CSMA/CA. Due to the concept of this medium access protocol, a shared channel cannot be occupied 100% of the time since in IEEE 802.11 the protocol delays every data transmission by a random backoff period. Although channel contention increases with increasing transmissions (and communicating nodes within reach), it is more likely that one node chooses a small backoff time during contention. Therefore, the channel is used more often – leading to an overall increase in packets per second when examining the summed number of transmitted packets from all traffic generators. Missing aggregation features of 802.11a lead to further aggravation concerning suboptimal channel usage. Due to larger confidence intervals for results obtained from the real experiment, the described effect is not as significant as in the simulation, but the overall trend is clearly visible. Note that for n traffic generators, each individual generator contributed roughly $1/n$ of the summed packets per second. Thus, the channel was shared equally. The averaged delays from all traffic generators can be obtained from Fig. 6(b). An increase of delay with a growing number of transmissions can be explained by increasing channel contention. Again, both simulation and real system reveal similar magnitudes for delay values as well. The results concerning delay of this scenario (where transmissions are limited to one hop in any case) strengthen the assumption that firmware/hardware queuing is responsible for the bad delay results in case of more than one wireless hop in the first scenario.

V. Conclusion and Future Work

In this article, we presented a framework for evaluation of wireless networks in simulation setups and real experiments from the same codebase. We demonstrated that our framework accomplishes its main goals – the twofold way of executing applications in both simulation and real system, without compromising real-world performance. The framework therefore enables evaluation in simulation (for experiments with large networks and different topologies with relatively low effort) and real system (for real-world influences and investigation of real parallelism). It consists of a component-based architecture and adapts differences in programming paradigms of simulators and real networking applications. Thanks to employing DPDK and OMNeT++, the framework resides completely in user space and gathers benefits from both simulator and user-space debugging capabilities – altogether resulting in a convenient and scalable platform for wireless networking experiments.

In our future work, we will enable the framework to fully exploit capabilities of high-speed wired NICs. They support features like multiple hardware queues, virtual functions and offloading techniques. This would enable scenarios like the evaluation of distributed NFV applications. Furthermore, existing 802.11 stacks in simulation frameworks can be enhanced and cross-checked against real hardware by utilizing our framework. We also plan to investigate state-of-the-art approaches concerning security protocols in WMNs, routing in WMNs, as well as multi-channel wireless communication, and aim to develop own solutions.

References

[1] Förderverein Freie Netzwerke e. V. Freifunk steht für freie Kommunikation in digitalen Datennetzen. Accessed: 21-Dec-2017. [Online]. Available: https://freifunk.net

[2] *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification*, IEEE Std. 802.11, 2016.

[3] G. Z. Papadopoulos, K. Kritsis, A. Gallais, P. Chatzimisios, and T. Noel, "Performance Evaluation Methods in Ad Hoc and Wireless Sensor Networks: A Literature Study," *IEEE Communications Magazine*, 2016.

[4] S. Uludag, T. Imboden, and K. Akkaya, "A Taxonomy and Evaluation for Developing 802.11-based Wireless Mesh Network Testbeds," *International Journal of Communication Systems*, vol. 25, no. 8, pp. 963–990, 2012.

[5] M. Backhaus, M. Theil, M. Rossberg, and G. Schaefer, "Towards a Flexible User Space Architecture for High-Performance IEEE 802.11 Processing," in *IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, Limassol, Cyprus, 2018, accepted and in press.

[6] C. P. Mayer and T. Gamer, "Integrating Real World Applications into OMNeT++," *Institute of Telematics, University of Karlsruhe, Karlsruhe, Germany, Tech. Rep. TM-2008-2*, 2008.

[7] R. Naumann, S. Dietzel, and B. Scheuermann, "Towards More Realistic Network Simulations: Leveraging the System-Call Barrier," in *Ad Hoc Networks*. Springer, 2017, pp. 180–191.

[8] T. Staub, R. Gantenbein, and T. Braun, "VirtualMesh: An Emulation Framework for Wireless Mesh and Ad Hoc Networks in OMNeT++," *SIMULATION*, vol. 87, no. 1-2, pp. 66–81, 2011.

[9] The MathWorks, Inc. Products and Services - MATLAB & Simulink. Accessed: 16-Mar-2018. [Online]. Available: https://www.mathworks.com/products.html

[10] ——. WLAN System Toolbox - MATLAB. Accessed: 16-Mar-2018. [Online]. Available: https://de.mathworks.com/products/wlan-system.html

[11] ns-3 project. ns-3. Accessed: 17-May-2018. [Online]. Available: https://www.nsnam.org/

[12] INET framework for the OMNeT++ discrete event simulator. Accessed: 18-May-2018. [Online]. Available: https://github.com/inet-framework/inet/

[13] J. O. Coplien, "Curiously Recurring Template Patterns," *C++ Rep.*, vol. 7, no. 2, pp. 24–27, Feb. 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=229227.229229

[14] J. Gettys, "Bufferbloat: Dark Buffers in the Internet," *IEEE Internet Computing*, vol. 15, no. 3, pp. 96–96, May 2011.

[15] A. Showail, K. Jamshaid, and B. Shihada, "Buffer Sizing in Wireless Networks: Challenges, Solutions, and Opportunities," *IEEE Communications Magazine*, vol. 54, no. 4, pp. 130–137, April 2016.