CommonCode: A Code-Reuse Platform for Wireless Network Experimentation

Junhee Lee, KAIST
Jinsung Lee, KAIST
Kyunghan Lee, North Carolina State University
Song Chong, KAIST

ABSTRACT

Experimentation of a wireless network protocol over the air is of significant interest. However, it is more rarely performed than simulation because of the difficulties in coding and debugging as well as lack of scalability and repeatability. In this article, the concept of a code-reuse platform making use of a simulation code directly for real experiments is revisited as an efficient and easy means of experimentation. Furthermore, an architecture and key components of an idealistic code-reuse platform are suggested, and then CommonCode, the most advanced code-reuse platform, is proposed. Through extensive simulations and experiments using CommonCode for the same codes, we demonstrate that CommonCode is valid and accurate in terms of protocol performance, and simultaneously fast and easy in terms of protocol development.

INTRODUCTION

How many wireless network protocols are experimented and simulated? It might be hard to provide exact numbers, but it is clear that the number of simulated protocols easily overwhelms the number of experimented protocols. According to our preliminary survey for papers published in IEEE INFOCOM 2001 on wireless network topics such as sensor networks, cognitive radio, WiFi, cellular networks, and delay-tolerant networks (DTNs), only 8 percent had experimented protocols, and for the papers on wireless network topics at INFOCOM 2011, 27 percent had experimented protocols. The percentage is drastically improved but still very limited. Why do researchers avoid performing an experiment? There are two main reasons. Simulation is:

- Rapid to implement
- Easy in validating the protocols

Especially, the convenience in coding and debugging as well as in execution with scalability and repeatability is attractive. Consequently, NS-2, QualNet, OPNET, GloMoSim, and several other simulators are very popular at this moment.

However, there is an unanswered question. Would the protocols be validated through simulation as much as they are expected to perform in reality? There are many case studies [1–3] appealing that there are various performance gaps, and most of them are induced by the incomplete consideration of complex interactions of multiple network layers in reality. For example, a wireless medium access control (MAC) protocol designed to assign fair data rates to wireless clients often fails to achieve its goal in reality due to the signal strength difference between clients, which makes one of them capture the channel all the time. This problem is typically not observed in simulation consisting of logical links. Thus, the protocol validated through simulation studies may not work as well in practice. If people in the research community wish to see more active realization of their suggested protocols, more efforts to fill the gap should be made by incorporating an experimental validation process within a protocol design loop. Still, there is skepticism about experimentation because it is considered to be expensive, time consuming, and labor intensive, so it is worth introducing a code-reuse platform that carries out both simulation and experimentation with the same code developed in the simulator.

A code-reuse platform is intended to provide an environment as easy as a simulation and as realistic as an experiment to network researchers. The fundamental idea of the code-reuse platform is to first implement protocols on a simulator and then extend this implementation into the real world environment *without any code modification*. The platform consists of three major components: a simulator, a hardware adaptor, and a real-time event scheduler. It allows researchers to program a code in the simulator for ease of debugging and validation. Then the code is distributed to multiple machines for the experiments, and the hardware adaptor generates real packets as instructed in the simulation code. Also, the MAC layer parameters as well as some physical (PHY) layer parameters (e.g., signal transmission power) set in the code are realized in the network interface card (NIC) through the hardware adaptor. Finally, the timing of events in both simulation and

0163-6804/12/$25.00 © 2012 IEEE

Platform	JiST/MobNet	NS-3	TWINE	CommonCode
Base simulator	JiST/SWAN	NS-2	GloMoSim (QualNet)	GloMoSim
Program language	C-based Parsec and Csim, Java	C++, OTcl	C	C
Scheduling granularity	Tens of milliseconds	Millisecond scale	Microsecond scale	Microsecond scale
MAC programmability	Unsupported	Unsupported	Emulation	Possible
PHY programmability	Unsupported	Unsupported	Emulation	Partial
MAC debugging	No	No	Yes	Yes

Table 1. *Comparison of the existing code-reuse platforms in the literature.*

experiment is judiciously handled by the event scheduler. The code-reuse platform provides a revolutionary opportunity to redesign a protocol for better understanding of practical environments with minimal effort except placing nodes in proper locations.

In this article, we survey the existing code-reuse platforms and propose the most advanced platform, *CommonCode*. The details of CommonCode, including its architecture, operations of its major components, and experimental validations, are followed throughout the article.

Code-Reuse Platforms

A realistic network protocol evaluation becomes more crucial to researchers who desire to either upgrade a currently deployed protocol or propose a new protocol in a clean slate manner. Irrespective of these two approaches, it is obvious that experimentation of the protocol in practice is the most convincing evaluation method. However, due to various inevitable factors such as scale, budget, space, and time, simulation has been accepted as an efficient substitute for experimentation. However, the simulation-based approach has spread more than necessary, even for a protocol that can and should be evaluated in a small-scale real network. To encourage implementation-based research, code-reuse platforms can be good candidates. In this section, we discuss the key components of an ideal code-reuse platform and then survey the existing platforms for their structural strengths and weaknesses regarding component-wise aspects.

An Ideal Code-Reuse Platform

An ideal code-reuse platform requires three key components:

- A hardware adaptor
- A real-time event scheduler
- A debugger

The essential role of each component is as follows. First, the ideal hardware adaptor should convert simulation packets into real packets on time with valid header formats depending on the type of protocols (e.g., TCP/IP format). Also, it should support a functionality to set MAC/PHY parameters instructed by the simulator to the NIC without incurring any overhead. Second, the perfect real-time event scheduler should be able to capture the detailed time schedule of simulation and convert it into an executable time schedule in the real hardware of an experiment entity (e.g., a Linux machine). It also needs to identify whether the system can handle the incoming event or not, and to reflect it into the converted time schedule, which means that non-executable events in the entity on time must be excluded and logged in a database as a potential source of performance loss. Third, the ideal debugger should offer researchers an environment in which a bug in the code does not cause a system crash (e.g., kernel failure). In other words, by preventing system failure in the early stage, an effort to recover the final state in an experiment from the scratch can be minimized.

Existing Code-Reuse Platforms

The earliest effort of this kind is JiST (JAVA in Simulation Time) combined with MobNet (Mobile Networking) [4]. JiST is a simulation platform originally designed to eliminate the inconvenience of platform dependency. Its simulation kernel runs on the JAVA environment, thus the portability of the simulator is guaranteed. On top of JiST, SWAN (Scalable Wireless Network Simulator) provides an easy programming environment to researchers. When JiST/SWAN is combined with MobNet, they become a code-reuse platform because MobNet converts a simulation packet to a real packet of an electronic signal by connecting JiST to the NICs. Using JAVA is a good approach to obtain independence of programming language and system, but this may result in performance degradation because of the difficulty of system-dependent optimization.

NS-3, the successor to the extremely prevalent simulator NS-2, also supports code-reuse, but the method is completely opposite. It aims at simulating real protocols implemented in the Linux kernel with far higher scalability. More interestingly, the real protocol packets are converted to generate simulation packets to interact with protocols developed in the simulator. We classify this methodology into a reverse code-reuse and do not focus on this direction in this article.

TWINE [5] is a high-fidelity emulation and simulation framework. It facilitates repeatable, flexible, and efficient experiments for a protocol

implementation and their verification in complex network scenarios. In the emulation framework, protocols implemented in practice can be performed while the physical and data link layers are simulated in the Linux kernel. The TWINE simulation framework enables packet exchanges between simulation entities and real network entities involved in experiments. TWINE is one of the most prominent platforms to closely approach the goal of a code-reuse platform. However, since TWINE uses a virtual physical world, it has imperfect modeling of physical layer events.

Existing code-reuse platforms have been designed for diverse original purposes. To be an ideal code-reuse platform, each needs a different direction of evolution, demanding a long-term agenda. Thus, to quickly reduce the gap between the idealism and the realism, we propose our own code-reuse platform, CommonCode. This is the first platform satisfying the aforementioned three key components. In the following section, we further describe the detailed implementation of CommonCode to clarify its capabilities and limitations.

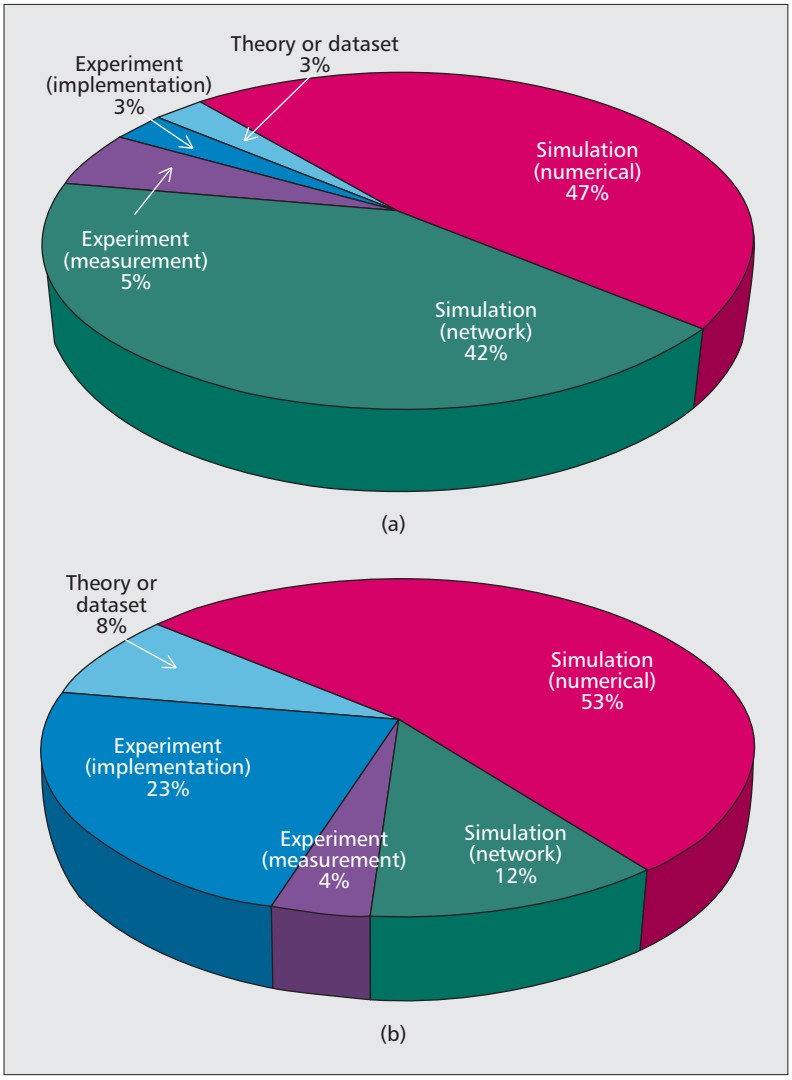

Figure 1. *Classification of papers on wireless topics published in IEEE INFOCOM 2001 and 2011 for their evaluation methods: a) INFOCOM 2001; b) INFOCOM 2011.*

COMMONCODE

In this section, we present more details of the components of a code-reuse platform by looking into the implementation of CommonCode whose conceptual architecture is illustrated in Fig. 2a. In addition, we clarify the challenges and limitations of the current implementation of CommonCode in detail.

HARDWARE ADAPTOR

A hardware adaptor has three major functions: packet conversion, MAC parameter control, and PHY parameter control. Among them, packet conversion is the core of a code-reuse platform, which converts a simulation packet to a real packet (with an electric signal) and vice versa by adding and removing protocol headers for various network protocols implemented in practical systems. Another important role of the hardware adaptor is interpreting MAC/PHY parameter values specified in the simulation code and setting them into the real NIC. This functionality can be realized by understanding the driver of a specific unit of network hardware. Most Atheros chipsets (compatible with the MadWiFi driver) are supported by the current implementation of CommonCode, which are not only popular but also known to provide the largest number of hardware control interfaces at the driver level. Consequently, CommonCode is capable of controlling MAC parameters such as contention window (CW) size and arbitration inter-frame space (AIFS) as well as PHY parameters such as transmission power (TxPower), link rate, and modulation scheme. The biggest benefit of controlling such parameters at the simulation level is the ability of nullifying IEEE 802.11 MAC implemented in the off-the-shelf NIC. Thereby, CommonCode is ready for MAC protocol design, simulation, and experimentation in an overlay manner via the help of an enhanced GloMoSim simulator.

From an architectural viewpoint, the hardware adaptor for CommonCode is unique in having been designed to support MAC protocol development, which is known to be very challenging. Note that a MAC protocol able to run over off-the-shelf devices should be in a form of an overlay on IEEE 802.11 carrier sense multiple access with collision avoidance (CSMA/CA) unless CommonCode is extended to support software defined radio (SDR) platforms (e.g., USRP and SORA). One of the main challenges is avoiding the usage of system application programming interfaces (APIs) such as `iwconfig` and `iwpriv` commands to control MAC/PHY parameters at every packet transmission. System APIs adopted in other code-reuse platforms (e.g., MobNet, NS-3) are convenient to use but unable to control packets within a timescale less than milliseconds, meaning that MAC-level parameter setting and events related to MAC operations are difficult to execute at the right time. CommonCode circumvents the problem by piggybacking the MAC/PHY control parameters into the adaptor header of each packet so that they can be performed directly in the driver.

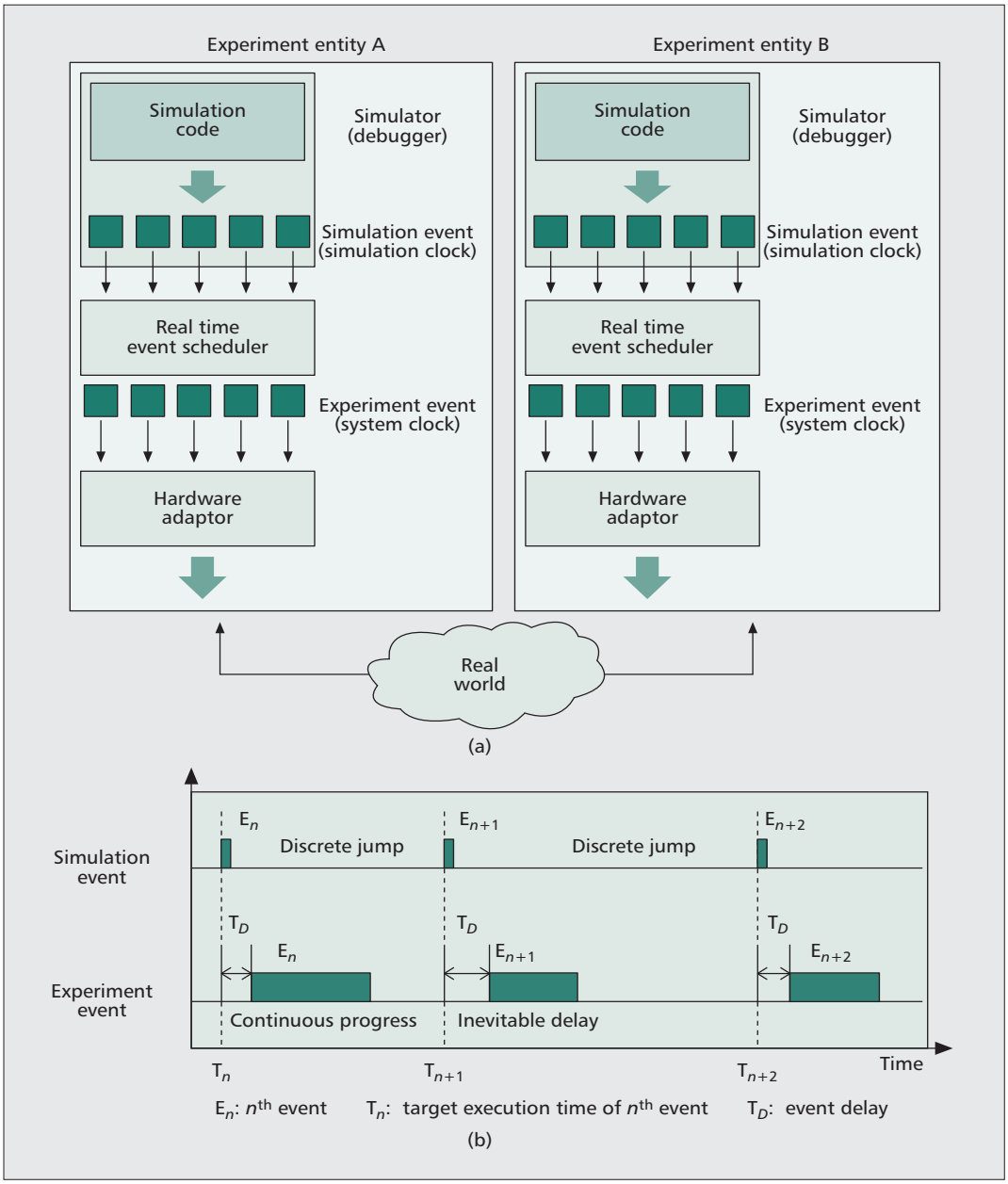

Figure 2. *Understanding CommonCode: a) the conceptual architecture and the event scheduler behavior of CommonCode; b) event scheduling delay.*

One of the most important features of CommonCode is its unique protocol debugging platform which is inherently provided by its code-reuse nature. It is expediting validation of newly developed protocols.

Real-Time Event Scheduler

The real-time event scheduler in CommonCode basically customizes the simulation event time into the real event time. Generally speaking, the conversion itself is a trivial operation. However, supporting MAC development, which requires scheduling of events on a tens of microseconds scale, makes the design more complex because a Linux machine is only capable of switching between processes at most in tens of milliseconds (i.e., the timescale known as *jiffy*). The real-time event scheduler in CommonCode consumes less time than that of kernel-level task switching, because all the events created in CommonCode are regarded as a single user application. Through this technique, the scheduler guarantees executing the events almost on time, as shown in Fig. 2b with minimal event delay T_D swaying up to 120 μs, which is negligible compared to the millisecond scale of the event delay in other code-reuse platforms. In all scenarios we tested, we cannot see any significant performance degradation due to the event delay, as shown in the next section.

Debugging Platform

One of the most important features of CommonCode is its unique protocol debugging platform, which is inherently provided by its code-reuse nature. It is expediting validation of newly developed protocols. The uniqueness of this platform comes from the fact that CommonCode can run a protocol in both simulation and experimental environments. More specifically, the technique comparing the results of CC-sim (simulation on CommonCode) and CC-exp (experiment on CommonCode, note that they completely share

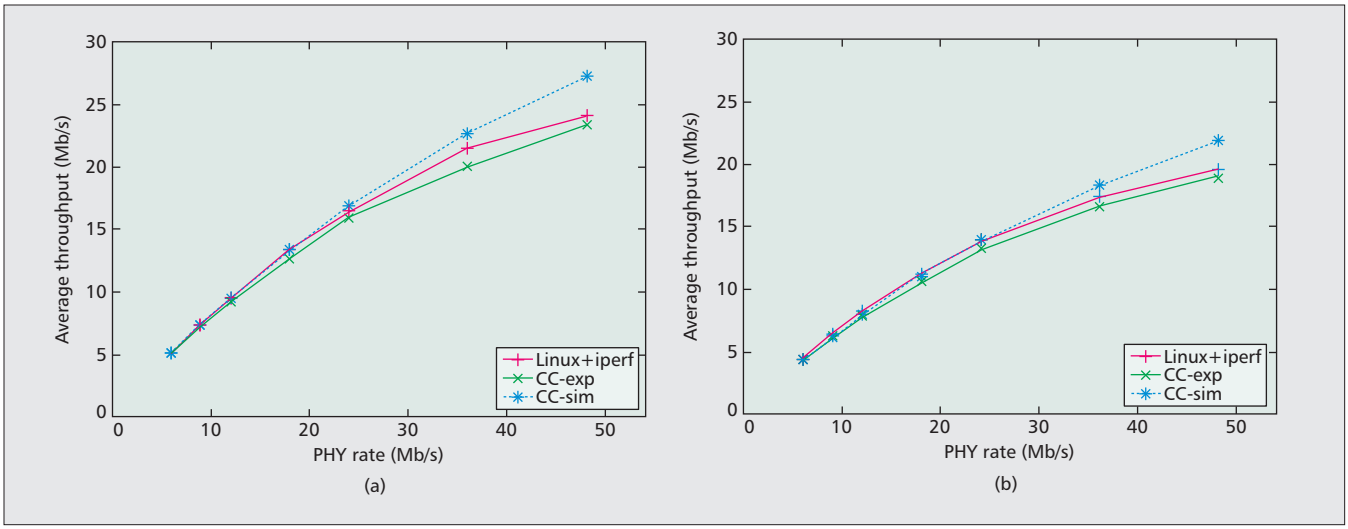

Figure 3. *1-hop flow's throughput comparison under various IEEE 802.11a rates: a) UDP performance; b) TCP performance.*

the same codes) greatly helps researchers isolate an error in their protocols, using the following idea. In a case where the results of CC-sim and CC-exp match but the results do not follow intuition, the logical design or implementation of network layers above the MAC layer may have an issue. Then CC-sim can be used for detailed debugging. On the other hand, in a case where the result of CC-sim is different from that of CC-exp, assumptions or abstractions made in the PHY layer may have an issue. Then CC-exp can work as a debugger. Compared to the protocol development relying on a simulator and a separate testbed, it is obvious that development using CommonCode is easier and faster. Through the evaluation section, we demonstrate how efficiently we can find potential issues in wireless network protocols by comparing results from CC-exp and CC-sim. Indeed, this ability of CommonCode assisted us a lot in implementing protocols for evaluations.

Evaluation

In this section, we demonstrate the validity of CommonCode by extensively comparing the performance of iperf on Linux (Linux+iperf), CC-sim, and CC-exp. Using a large-scale testbed, we verify our platform by experimenting complex network scenarios and further implementing a well-known cross-layer heuristic algorithm.

Evaluation Setup

Our experimental testbed is composed of commercial netbooks (1.66 GHz CPU and 1Gbyte RAM), each of which is equipped with an IEEE 802.11a NIC (Atheros AR5006 chipset). We use Linux 2.6.31 and MadWiFi v0.9.4 for the NIC driver, with some modifications described in the previous section. Throughout evaluations, we use the same parameters for CC-sim and CC-exp, which are consistent with the default setting in iperf v2.0.4. Specifically, UDP packet size of 1470 bytes is used, and TCP MSS, TCP send buffer, and TCP receive buffer are chosen as 1448 bytes, 16 kbytes, and 85.3 kbytes, respectively. We use TCP Reno, but other versions of TCP are also verified to have similar behaviors in test scenarios. All results are averaged from 10 runs of 60 s each. Moreover, RTS/CTS exchange and rate adaptation are disabled throughout the experiments. Note that the CPU usage of CC-exp is shown to be comparable to that of Linux+iperf, and the memory usage of CC-exp itself is verified to be very small, excluding the size of the storage buffer generated inside CommonCode.

Performance in a Controlled Setting

We measure the throughput performance of CommonCode to characterize its efficiency of operations in several controlled scenarios. To minimize the impact of variations in the wireless channel such as time-varying interference and fading, we deploy all nodes in close proximity (i.e., all nodes are fully connected) so that they persistently interfere with each other. In the first scenario, a sender-receiver pair is 1 m apart and uses IEEE 802.11a operating in the 5.805 GHz band with various link rates. For each link rate, we observe the throughput from a TCP source as well as a UDP source designed to saturate the link rate.

Figure 3 shows the throughput curve for both UDP and TCP traffic under various IEEE 802.11a link rates. Overall, CommonCode produces very similar performance to Linux+iperf without losing a noticeable amount of performance. Interestingly, at lower rates, the performance of CommonCode is very close to that of Linux+iperf, while at higher rates, the performance of CC-sim is slightly higher than the others. This can be explained as follows. Since CommonCode has additional overhead due to the adaptor header, it gets a bit less throughput than Linux+iperf. Despite a small loss of performance, CC-exp is very accurate in estimating the performance of experiments. The result of CC-sim well demonstrates the benefit of CC-exp. In most IEEE 802.11 chipsets, control packets such as acknowledgment, request to send, and clear to send (ACK/RTS/CTS) are designed to be sent using the basic data rate (e.g., 6 Mb/s in 802.11a). CC-exp simply follows this design as it also uses

the chipset, but CC-sim sending such control packets with the same data rate of data packets shows a mismatch in performance by inflating the throughput. More sophisticated modeling in the simulator might be able to reduce this performance gap between experiment and simulation. However, given that perfect modeling is nearly unachievable, CC-exp is recommended for realistic performance evaluations.

To further investigate CommonCode performance, we conduct two other experiments using IEEE 802.11a basic rate, 6 Mb/s. In one scenario, we measure the performance when a single flow is transmitted via different numbers of hops, and in another scenario, we measure the performance when several single-hop flows contend for a channel within a network. For the first scenario, Figs. 4a and 4b show the throughput of CommonCode against Linux+iperf for both UDP and TCP traffic. Irrespective of the number of hops and traffic types, we see very close match between CommonCode and Linux+iperf. In the second scenario, Fig. 4c presents the normalized TCP performance of CommonCode against Linux+iperf, indicating that there is some level of performance loss of CommonCode over Linux+iperf. Simply, the gap is due to the additional overhead of adaptor headers. However, CC-sim experiences more throughput loss than CC-exp as the number of competing links increases. We verify that the reason for performance loss is the imperfect modeling of the packet *capture effect* [6] in simulators. In CC-sim, a collision is always treated as a failed transmission, and there is no capture effect. On the other hand, in experiments (i.e., CC-exp and Linux+iperf), the collided packets can be decoded with higher signal strength at the receiver.

Performance in Uncontrolled Setting

Now, we evaluate the impact of wireless medium and physical layer interaction using an indoor testbed shown in Fig. 5a. We experiment with several scenarios by changing the number of injected flows on the testbed. Since CC-sim is not able to perfectly model details of the wireless channel such as interference and physical capture effect, we only compare the performance of Linux+iperf and CC-exp. Figures 5b–e show the throughput comparison between them. For better readability of graphs, the throughput values from different flow indexes are arranged in increasing order. In all cases we verify that Linux+iperf and CC-exp show almost the same trends in throughput distribution to different flows as well as their aggregate throughput. More important, CC-exp is successful in capturing the throughput degradation in certain flows due to collisions and complex interferences induced by the physical environment of the topology, as if Linux+iperf is affected by the environment.

Furthermore, we demonstrate the usefulness of our platform by implementing a queue-based heuristic link scheduling algorithm, *DiffQ* [7], which was originally developed as a kernel module tailored for Linux kernel 2.6.18. In scheduling multiple links in proximity, DiffQ prioritizes links based on queue differential. Thus, it can schedule queued packets in a fairer manner even

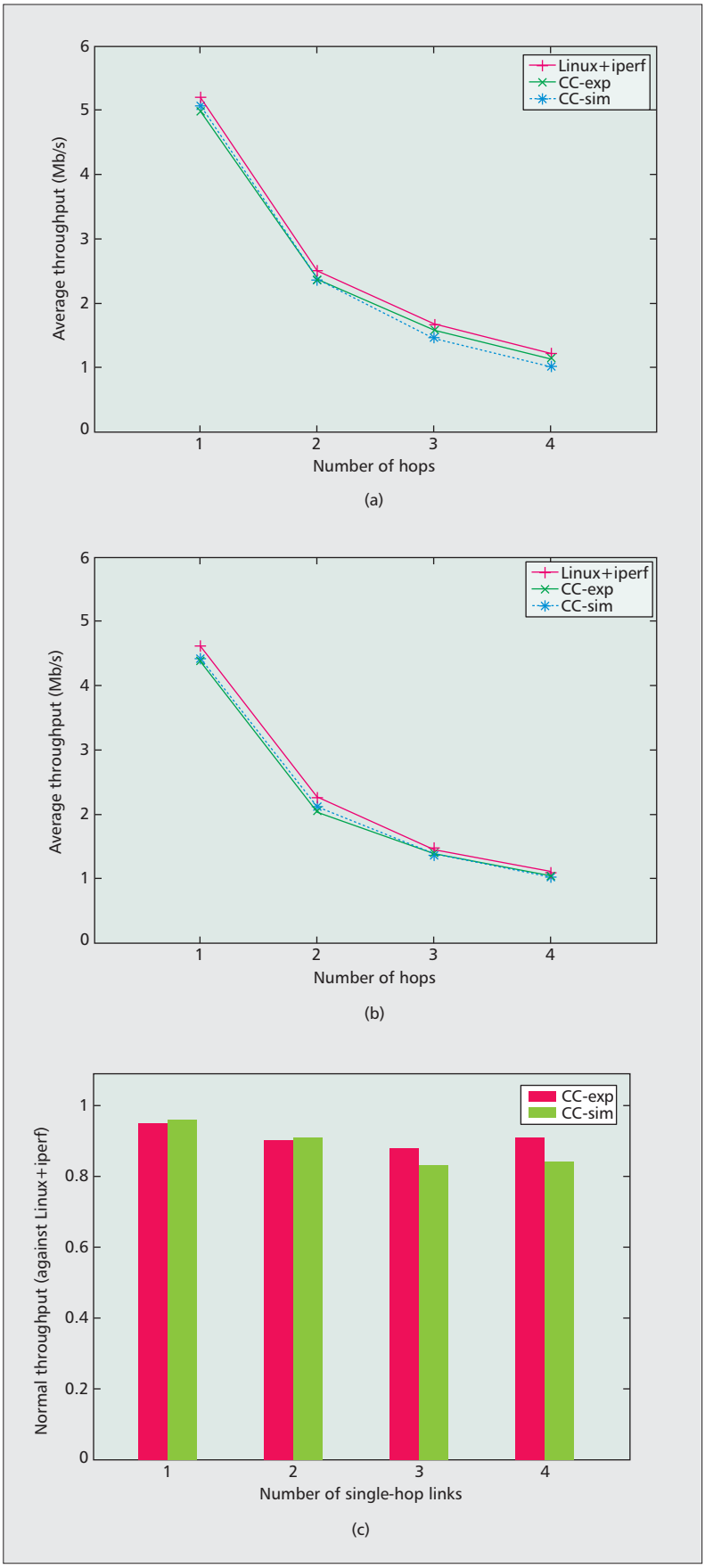

Figure 4. *Throughput comparison using different settings at 6 Mb/s: a) UDP performance over different numbers of hops; b) TCP performance over different numbers of hops; c) normalized TCP performance over different numbers of single-hop flows.*

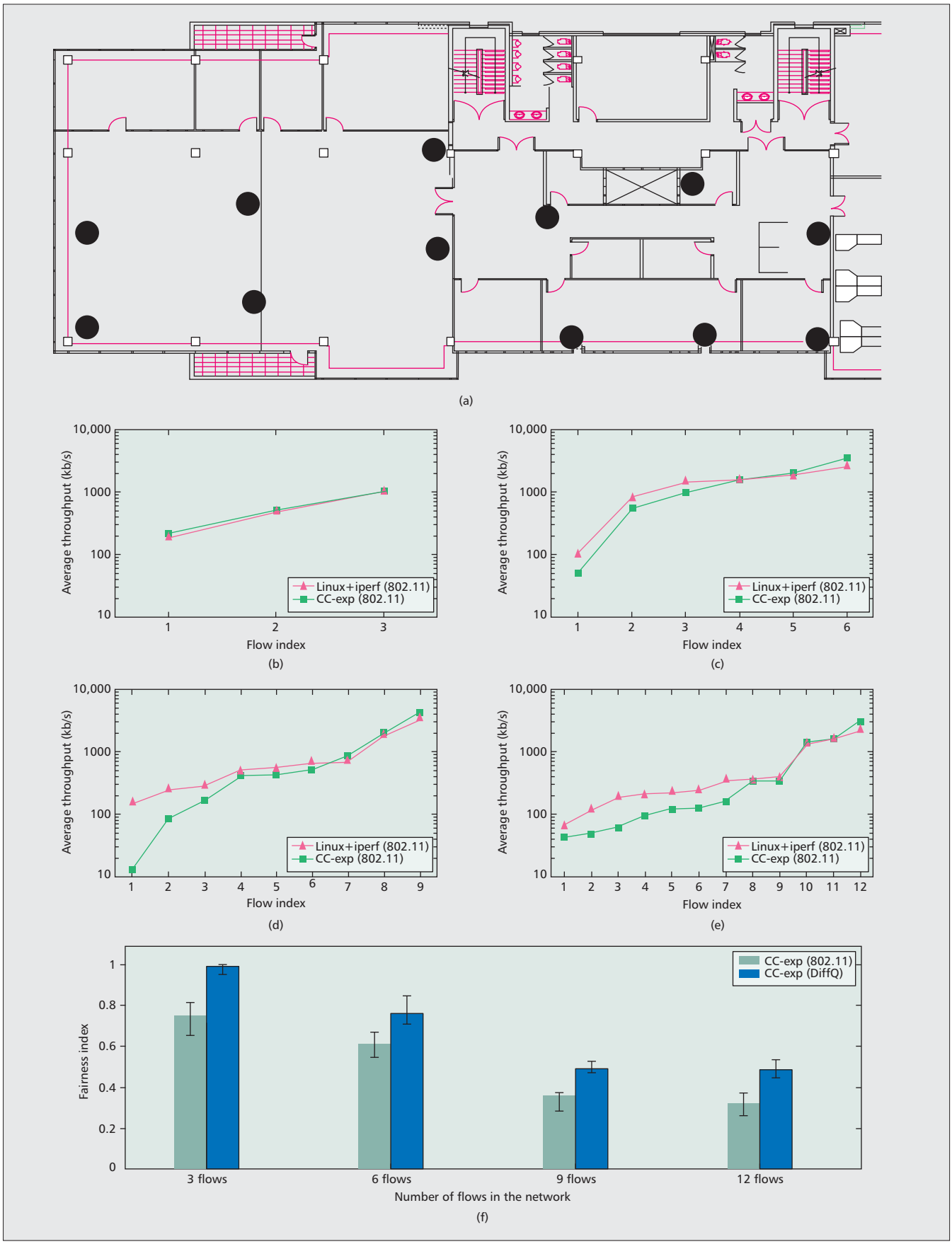

Figure 5. *Performance comparison on an indoor testbed consisting of 12 nodes: a) testbed map: 12 nodes denoted by black circles are distributed in a floor of a building whose space is about 40 m × 20 m; b) throughput comparison for three flows; c) throughput comparison for six flows; d) throughput comparison for nine flows; e) throughput comparison for 12 flows; f) fairness comparison between 802.11 and DiffQ, both of which are implemented over CommonCode.*

under asymmetric links by giving more transmission chances to less served links while 802.11 schedules links irrespective scheduling histories.

We implemented the core of the DiffQ algorithm as a module in CommonCode. The implementation took only about a month, which is much shorter in time comparing to its original implementation in both the kernel and driver. Using the testbed shown in Fig. 5a, we evaluate the performance of DiffQ and 802.11 under CommonCode. For the comparison of throughput fairness, we inject several single-hop flows and calculate Jain's fairness index. Figure 5f compares the fairness index of two protocols. As expected by the nature of DiffQ, the fairness of DiffQ is much higher than that of 802.11 in all tested scenarios. From the results, we believe that CommonCode provides a much easier way to experiment with large-scale networks accurately.

As expected by the nature of DiffQ, the fairness of DiffQ is much higher than that of 802.11 in all tested scenarios. From the results, we believe that CommonCode provides a much easier way to experiment with large-scale networks accurately.

Conclusion

In this article, we survey the existing code-reuse platforms and identify the idealistic architecture of such platforms. Also, we propose CommonCode, the most advanced code-reuse platform, with a detailed description of the implementation of its key components. Through extensive simulation and experiment, CommonCode is demonstrated to be valid, efficient, and easy enough to use. It is also encouraging that recent developments of several cross-layer transport/routing/MAC protocols are successfully completed within much shorter periods than typical kernel-level development.

While we have been completing this platform, we provided its primitive version to develop a theory-based algorithm using conventional 802.11 hardware. From the recent work [8, 9], we remark that our proposed platform enables a faster and easier transition from theory to practice despite limited functionalities. For a full level of PHY/MAC programmability, we can integrate an SDR adaptor to CommonCode, which is scheduled as our future work.

Acknowledgment

This research was supported by the Korea Communications Commission (KCC) under the R&D program supervised by the Korea Communications Agency (KCA) (KCA-2011-09913-04005) and the Ministry of Knowledge Economy (MKE), Korea, under the Information Technology Research Center (ITRC) support program supervised by the National IT Industry Promotion Agency (NIPA) (NIPA-2011-(C1090-1111-0004)).

References

[1] D. Kotz *et al.*, "Experimental Evaluation of Wireless Simulation Assumptions," *Proc. ACM MSWiM*, 2004.
[2] R. S. Gray *et al.*, "Outdoor Experimental Comparison of Four Ad Hoc Routing Algorithms," *Proc. ACM MSWiM*, 2004.
[3] S. Ivanov, A. Herms, and G. Lukas, "Experimental Validation of the ns-2 Wireless Model Using Simulation, Emulation, and Real Network," *Proc. WMAN*, 2007.
[4] T. Krop *et al.*, "JiST/MobNet: Combined Simulation, Emulation, and Real-World Testbed for Ad hoc Networks," *Proc. ACM WinTECH*, 2007.
[5] J. Zhou, Z. Ji, and R. Bagrodia, "TWINE: A Hybrid Emulation Testbed for Wireless Networks and Applications," *Proc. IEEE INFOCOM*, 2006.
[6] J. Lee *et al.*, "An Experimental Study on the Capture Effect in 802.11a Networks," *Proc. ACM WinTECH*, 2007.
[7] A. Warrier *et al.*, "DiffQ: Practical Differential Backlog Congestion Control for Wireless Networks," *Proc. IEEE INFOCOM*, 2009.
[8] J. Lee *et al.*, "Implementing Utility-Optimal CSMA," *Proc. Allerton Conf.*, 2009.
[9] B. Nardelli *et al.*, "Experimental evaluation of Optimal CSMA," *Proc. IEEE INFOCOM*, 2011.

Biographies

JUNHEE LEE received his B.S. in electrical engineering from Kyungpook National University and his M.S. in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 1998 and 2000, respectively. He is currently a Ph.D. candidate in the Department of Electrical Engineering at KAIST. His research interests include network experimentation platforms, multihop wireless networks, and their architecture design and implementation.

JINSUNG LEE received his B.S. in electrical engineering from KAIST in 2003. He is currently a Ph.D. candidate in the Department of Electrical Engineering of the same university. His research interests include cross-layer optimization, multihop wireless networks, and their protocol design and implementation.

KYUNGHAN LEE [S'07, A'10] received his B.S., M.S. and Ph.D. degrees in electrical engineering and computer science from KAIST in 2002, 2004, and 2009, respectively. He is currently a senior research scholar in the Department of Computer Science at North Carolina State University. His research interests are in the areas of human mobility, delay tolerant networks, context-aware services and applications, mobile systems, and protocol design.

SONG CHONG [M'93] received B.S. and M.S. degrees in control and instrumentation engineering from Seoul National University, Korea, in 1988 and 1990, respectively, and his Ph.D. degree in electrical and computer engineering from the University of Texas at Austin in 1995. Since March 2000, he has been with the Department of Electrical Engineering, KAIST, where he is a professor and was head of the Communications and Computing Group of the department. Prior to joining KAIST, he was with the Performance Analysis Department, AT&T Bell Laboratories, New Jersey, as a member of technical staff. His current research interests include wireless networks, future Internet, and human mobility characterization and its applications to mobile networking. He has published more than 100 papers in international journals and conferences. He is an Editor of *Computer Communications* and the *Journal of Communications and Networks*. He has served on the Technical Program Committees of a number of leading international conferences including IEEE INFOCOM and ACM CoNEXT. He serves on the Steering Committee of WiOpt and was the General Chair of WiOpt '09. He is currently the Chair of Wireless Working Group of the Future Internet Forum of Korea and the Vice President of the Information and Communication Society of Korea.