RealProct: Reliable Protocol Conformance Testing with Real Nodes for Wireless Sensor Networks

Junjie Xiong[1] Edith C.-H. Ngai[2] Yangfan Zhou[1] Michael R. Lyu [1,3]
[1]Dept. of Computer Science & Engineering, The Chinese Univ. of Hong Kong, Shatin, Hong Kong
[2]Dept. of Information Technology, Uppsala University, Sweden
[3]School of Computer Science, National University of Defense Technology, Changsha, China.

***Abstract*—Despite the various applications of wireless sensor network (WSN), experiences from real WSN deployments show that protocol implementations in sensor nodes are susceptible to software failures, which may cause network failures or even breakdown. Pre-deployment protocol conformance testing is essential to ensure reliable communications for WSNs. Unfortunately, existing solutions with simulators cannot test the exact hardware and implementation environment as real sensors, whereas testbeds are expensive and limited to small-scale networks and topologies.**

In this paper, we present *RealProct*, a novel and reliable framework for testing protocol implementations against their specifications in WSNs. RealProct utilizes real sensors for protocol conformance testing to ensure that the results are close to the real deployment. Using different techniques from those in simulations and real deployments, RealProct virtualizes a large network with any topology and generate non-deterministic events using only a small number of sensors to provide flexibility and to reduce the cost. The framework is carefully designed to support efficient testing in resource-limited sensors. Moreover, test execution and verdict are optimized to minimize the number of runs, while guaranteeing satisfactory false positive and false negative rates. We implement RealProct and test it with the 1IP TCP/IP protocol stack and a routing protocol developed for WSNs in Contiki-2.4. The results demonstrate the effectiveness of RealProct by detecting several new bugs and all previously discovered bugs in various versions of the μIP TCP/IP protocol stack.

I. Introduction

Protocols play an important role in wireless sensor networks (WSNs) for reliable communication and coordination [1] [2]. However, experiences from real deployments show that protocol implementations in sensor nodes are susceptible to software failures despite pre-deployment testing [3] [1] [4]. Even though specifications are available for standardized protocols, it is still very hard to assure flawless protocol implementations by software developers. Misinterpretation of the specification or software bugs in implementation could both cause communication failures [2] [4], and even breakdown the whole network. It can be very expensive and difficult to fix the bugs after deployment [5] [6]. Although simulations are applied to detect software bugs for WSNs before deployment [5] [6], it is widely believed that simulations may miss bugs that only expose in real hardware and execution environment. Although testbeds are considered for pre-deployment testing, they are expensive and limited to small-scale networks. In addition, most of the existing testbeds are designed for network performance evaluation rather than software bug detection [7]. To uncover as many problems as possible, the best method is to test with large-scale real WSNs which is, however, too expensive.

To settle the above problem and achieve an effective tradeoff between large-scale real deployment and simulation, we present **RealProct** that use a few low-cost real sensor nodes to mimic large-scale real WSNs and then perform protocol conformance testing for WSNs reliably and cost-effectively. With RealProct, the testing is the closest to the real deployment while the cost is kept at a low level. RealProct is motivated by protocol conformance testing (PCT), an authoritative method to check the consistency between a protocol implementation and its specification [8]. To test a protocol with RealProct, the tester only needs to design a set of abstract test cases according to the protocol specification. Although PCT have been applied for Internet, 3G network, WiMAX network [9] [10], its applications for WSNs have not yet been investigated.

We observe that WSNs that are dramatically different from the traditional networks. Moreover, RealProct is also very distinct from the simulations and real deployments. As a result, several major research challenges exist. First, sensor nodes are resource-constrained devices, so they are unable to provide the same facilities and operations as the standardized PCT test system. For example, the sensor node is incapable of storing all the test cases in its limited memory. Second, a sensor node in the PCT test system is more difficult to be controlled than a personal computer (PC). A sensor node usually does not provide proper user interface, so operation commands are sent to the node mainly relying on the serial ports connecting to the computer. Third, the volatile wireless environment in WSNs will result in random packet loss [1]. The loss may trigger the potential protocol bugs and results in instability and unreliability in our tests, such as false positive and false negative errors [11]. Finally, RealProct only employs a few real sensor nodes, and hence it seems that it is unable to test the protocol with various topologies and events as the simulations and high-cost real deployments.

To attack the challenges, our major contributions are as follows:

- We propose RealProct as a generic and cost-effective architecture for protocol conformance testing in WSNs that makes use of only two real sensor nodes and a personal computer (PC). The PC helps storing all the test cases which will be handed over to the sensor nodes in real time. The framework enforces abstracting the protocol interfaces to build up a library for concise control of the sensor nodes through serial ports.
- Three different techniques (not utilized in simulations and real deployments) are proposed to make RealProct practical and reliable for WSN protocol conformance testing: topology virtualization, event virtualization, and dynamic test execution. The topology virtualization technique imitates WSNs with different structures using a few sensor nodes, while the event virtualization generates non-deterministic events. Dynamic test execution can tackle the inaccuracy problem caused by non-deterministic events when generating the test verdict.
- We provide case studies with real experiments to prove the feasibility of RealProct. RealProct effectively discovers two new bugs verified by the developers, and identifies all the previously detected bugs in the μTCP/IP stack [12] [13] of Contiki-2.4 [14]. It also validates the Rime mesh routing protocol in Contiki-2.4. The source code for finding the bugs is available online[1].

The remainder of this paper is organized as follows. Section II discusses the related work. Section III presents the background of PCT. We design the framework of RealProct for protocol conformance testing in WSNs in Section IV, and propose three novel techniques to make RealProct practical for WSNs in Section V. After discussing the generality of RealProct in Section VI, we evaluate its feasibility with two case studies in Section VII. Finally, Section VIII concludes the paper.

II. Related Work

Existing approaches for testing and debugging WSN protocols fall into three main categories: program analysis tool and simulator (testing before deployment), real deployment (testing after deployment), and debugger. We consider testing as the technique for detecting failures or abnormal behaviors, while debugging as the process for diagnosing the precise nature of a known error [15]. Only after abnormal behaviors are detected in testing, debugging will be triggered.

Program analysis tools and simulators are very useful in detecting errors and verifying algorithms of the protocols. Safe TinyOS [16] is compile-time program analysis tool that can avoid memory corruption before program execution. FSMGen [17] derives FSM from TinyOS [18] application so as to help the programmer detect the software errors at a system level. Well-known simulation tools include ns-2 [19], a general network simulator, TOSSIM [20], a simulator for TinyOS WSNs, and Avrora [21], a cycle-accurate instruction-level simulator. Several tools developed for finding software bugs in WSNs are based on simulation, such as T-Check [5] and KleeNet [6]. For example, T-Check [5] is an event-driven simulator based on TOSSIM and Safe TinyOS, while KleeNet [6] is based on the low-level symbolic virtual machine KLEE [22]. Given the fact that nodes in a simulation environment and real sensor nodes are different, protocol execution in simulations may not be able to discover all software problems that will occur in real sensor nodes.

Different from simulations, testing during protocol execution in real sensor nodes is crucial in detecting software bugs. Sympathy [23] actively collects run-time status from sensor nodes, such as routing table and flow information, and detects possible faults by analyzing the information. To save energy, PassiveDiagnosis [7] passively records the logs from real deployment by piggybacking the logs in regular data packets. Although this run-time testing method can detect real-time failures in real deployment, it is costly and very difficult to detect and fix errors after deployment. Similarly, testbeds could be applied for testing in real nodes, but they are expensive and limited to small-scale networks [7]. Moreover, existing testbeds are often designed for network performance evaluation rather than protocol conformance testing [7]. Different from the above work, RealProct uses a few real sensor nodes to perform protocol conformance test, which can detect bugs on real nodes efficiently before large-scale deployment.

A number of debuggers have been developed for WSNs. Clairvoyant [24] is a comprehensive source-level debugger that provides standard debugging commands including break, step, watch, and so on. Since it can only debug a sensor node instead of a network, it cannot handle distributed bugs. Although some debugging techniques can detect abnormal behaviors, they mainly focus on locating the root cause of the software bugs. DustMiner [25] diagnoses the root cause of the exposed anomalous behavior by sequence extraction and frequent pattern analysis.Sentomist [4] and tracepoints [26] are both automated tools for debugging WSN applications. The former is built based on simulation and is especially effective for identifying transient bugs. The latter inserts checkpoints in the applications to allow run-time debugging. Since debuggers are designed for locating the bugs after the bugs are detected by testing, they only give a very detailed investigation on a small part of the codes for locating specific bugs. In this work, we focus on protocol conformance testing rather than debugging, which allows us to detect general implementation problems for WSN protocols.

[1] http://www.cse.cuhk.edu.hk/~jjxiong/testcases/testcase.htm

III. Protocol Conformance Testing

Due to programming bugs or misunderstanding of the standard specifications, the protocol implementation always differs from the expected, hindering the interoperability between systems implemented by different manufacturers [8] [27]. Hence, testing is necessary. Protocol conformance testing (PCT) is an authoritative way to check the consistency between the any protocol and its implementation [8]. As functional (black-box) testing, PCT is popular and influential not only in traditional wired networks [27], but also in wireless networks, such as WiMAX network [9] and 3G network [10]. Despite PCT's wide application, it has not yet benefited WSNs due to a number of challenges discussed previously.

A. PCT Process

The PCT process is shown in Fig. 1, in which the protocol implementation to be tested is called **Implementation Under Test** (IUT). Its first phase is test generation, in which abstract test cases for the IUT are designed based on the protocol specification. Each test case can only have one target, i.e., testing a particular function of the protocol. A collection of test cases can cover many aspects of the protocol, and they form a test suite. Formal methods, such as FSM and extended FSM [27], are employed for deriving the abstract test suite.

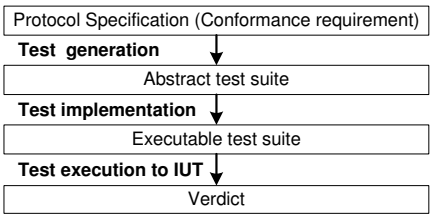

Figure 1. PCT testing process.

The second phase, test implementation, transforms the abstract test cases into executable test cases that can be executed in a real test system. The last phase is test execution, in which the implemented test cases are executed against the IUT and the resulting behavior of the IUT is observed. This leads to the verdict about the conformance of the IUT with respect to the protocol specification. After PCT, we can rerun those failed test cases to diagnose the problems so as to improve the protocol implementation.

B. PCT architecture

There are mainly four classical architectures for PCT: local, distributed, coordinated, and remote architectures. Since their difference lies in the component organization rather than in the component number, we only discuss the PCT distributed architecture as shown in the biggest rectangle (not including the 4 blue items) in Fig. 2. The architecture is composed of two entities: the **System Under Test** (SUT) and the **tester**. In traditional Internet protocol conformance testing, both SUT and tester could be a PC.

The points where the tester controls and observes the IUT are called the Points of Control and Observation (PCO). For IUT at layer N, its specification identifies the behavior of a protocol entity at the upper and lower access points (interfaces) of the protocol. Hence the ideal PCOs to test the IUT are these two access points. The part that is connected through the upper access point is called the Upper Tester (UT). Similarly, the part that is connected to the IUT through the lower access point is called the Lower Tester (LT). The service provider supports the UT, IUT, and LT implementation.

The tester contains only LT while the SUT contains two logical components: UT and the IUT. The LT is consisted of executable test cases, and it is a peer entity of the combination of the UT and the IUT. Each test case execution stimulates the IUT, and the response of the IUT is observed at PCOs. Test results are obtained by comparing the observations with the expected behaviors indicated in the specifications. In other words, the LT provides different kinds of inputs to IUT, and the outputs of the IUT are compared with the correct behaviors as specified to arrive at a verdict. When testing IUT, we focus on the operation of the protocol being tested and consider that the other parts behave correctly. In this way, the failed test cases can pinpoint the problems in protocol implementation. In addition, the test coordination helps execute the tests, which includes maintaining synchronization between the SUT and tester and controlling the execution flow.

IV. Design of RealProct Framework

The above PCT design is a general software engineering method for testing protocol implementation against specifications. However, none of the above four PCT architectures fit WSNs due to the limited resources of sensor nodes and their volatile working environment. For this reason, our designed framework for PCT is different from the previous general one in the 4 blue items as shown in Fig. 2. To unveil as much abnormal behaviors of a protocol implementation in WSNs before deployment, we utilize two real sensor nodes to work as the tester and the SUT respectively in a distributed PCT architecture. We explain the unique features of our design as follows.

1) Number of real sensor nodes: We utilized only two real sensor nodes rather than 3 or dozens of real sensor nodes for three reasons. First, the general PCT architecture in the ISO9646 standard [8] employs only two entities as the SUT and the tester respectively. It means that even the tester is only composed of 1 sensor node, it can still test the protocols adequately. Second, although we can utilize 3 or more real nodes to function as the tester, it is much more difficult to control and coordinate larger number of independent devices in the required testing process. Third,

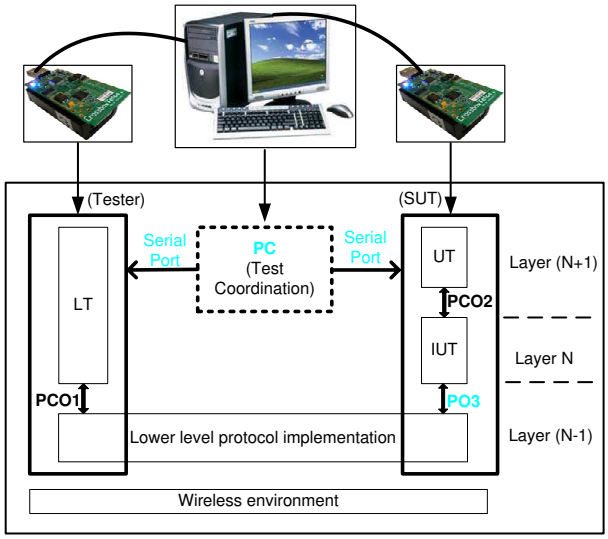

Figure 2. RealProct framework.

our virtualization technique allows us to virtualize more than 1 sensor nodes with only 1 real sensor node, and hence the SUT can hardly distinguish 1 real sensor nodes from more real sensor nodes.

2) Coordination: We suggest test coordination to be functioned by a PC, rather than by the tester or the SUT, due to the limited memory and computation power of the sensor nodes. Compared with the sensor nodes, PC is a much more powerful device to coordinate the tests. We connect the PC with the two sensor nodes with serial ports, through which the PC communicates with the sensor nodes and supply them with power. Initially, all the test cases are stored in the PC, and each one is downloaded to the tester for execution one by one.

In addition, the PC maintains test synchronization and controls sequence of actions between the tester and the SUT. Since we cannot manipulate the system to be tested (SUT) directly, we control it passively through the tester which will be installed with all test cases. The details on how we control it will be discussed in the next section. Furthermore, the PC observes and saves the test execution logs through the serial ports. These logs are very important for generating the test verdict and performing bug analysis afterwards. In practice, we do not save the test execution results in the sensor nodes due to their limited memory. Moreover, saving logs in the sensor node may alter or even deteriorate the performance of the IUT. If we store all the logs in the sensor nodes, we will not know the real-time results of the testing unless the sensor nodes send the logs to the PC. Instead, we suggest the PC to get real-time results from sensor nodes through the serial port and save the logs locally.

3) Resource management: The design of the LT and the UT are modified to better suit the resource-constrained WSNs. Unlike the traditional PCT framework, the LT and the UT are simplified for easier control of the tests. The LT is a suite of executable test cases based on FSM. Each test case in the LT now only targets at one aspect of the IUT. To make the test case simpler, we apply virtualization techniques to imitate a large network by controlling only two nodes rather than managing a large number of nodes. The UT should be simple, but still be capable to drive the IUT to send and receive packets. The log generated from the LT and the UT is mainly about packets in order to relieve the load of the tester and the SUT.

4) PCO optimization: We further optimize the PCOs for the testing framework and add logging module for the PCOs. In traditional PCT architectures, one PCO is in the tester and the other is the SUT. In contrast, our framework adds a point of observation (PO3) in the SUT as shown in Fig. 2. Is it redundant to add such a point? The answer is no. The traditional distributed PCT architecture assumes that traditionally both the PCO1 and PO3 observe the same phenomenon. However, this assumption does not hold due to the volatile wireless environment of WSNs. The LT can control and stimulate the IUT at PCO1, but it cannot make sure that the packet it sends will arrive at PO3 without loss, corruption, or duplication. To avoid false positive (FP) and false negative (FN) errors in test, we double-check the packet content at PO3.

V. RealProct Techniques

RealProct provides a generic framework for testing the implementation of a wide range of protocols in WSNs. Although it is straightforward to write test cases based on the specifications, it is difficult for RealProct to fulfill the testing requirements for different protocols using only limited number of sensor nodes in practice. To address this limitation, we propose two novel techniques, *topology virtualization* and *event virtualization*, to aid the LT design in order to imitate WSNs with various topologies and events in our 2-node framework. Apart from that, testing with real nodes also experiences random packet loss, which might cause FP and FN errors in test verdict. We thus design a *dynamic test execution* algorithm to reduce the FP and FN error rates down to a required level.

A. Topology virtualization

Because the simulations and real deployments are able to verify the protocol functionality various topologies, they do not need to virtualize topology at all. However, for RealProct that tests with limited number of nodes, topology virtualization is necessary to provide higher test case granularity. Unlike simulation, virtualization in our framework utilizes real hardware. The technique can virtualize larger scale network with limited real nodes in a way that the SUT can hardly distinguish them. The idea is that no matter in what kinds of topology, the SUT can only recognize its environment from the packets it receives. More importantly,

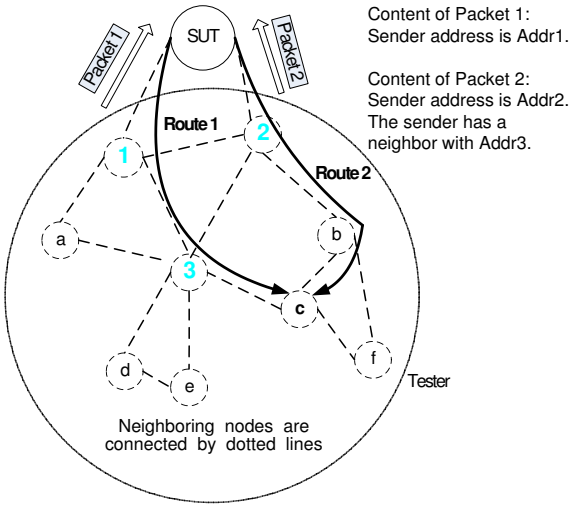

Figure 3. A virtualized topology.

it can only receive packets from the nodes that are placed within its reception range, i.e., 1-hop neighbor nodes. In practice, the amount of sensor nodes that are placed within a sensor node's reception range is limited. Hence, it is not difficult for the tester to virtualize all the 1-hop neighbors of the SUT. It seems that the tester can only virtualize 1-hop topologies, but not multi-hop topologies apparently. However, this is not true. We will show how the packets from nodes farther away can also be imitated and transmitted to the SUT by topology virtualization.

Specifically, given two real sensor nodes: the SUT and tester, and the latter can virtualize for the former a topology composed of node 1, node 2, and node 3 as shown in the upper portion of Fig. 3. The tester sends two routing packets to the SUT. Since the SUT can receive packet 1 and packet 2 directly, it regards the nodes that send these packets as its 1-hop neighbors (node 1 and node 2 here). In addition, packet 2 tells the SUT that it has a neighbor node 3. Since the SUT has not received any packets from node 3 directly, it adds node 3 as its 2-hop neighbor. As a result, the SUT experiences as if it has two 1-hop neighbors and one 2-hop neighbor in a two-level tree topology.

In real deployment of this topology, the packet receptions and topology discovery for the SUT are just the same as those in our virtualized topology. In other words, as long as the SUT receives the same packets as those in a real topology, the virtualized topology for the SUT is indistinguishable from the actual one. With topology virtualization, we can effectively reduce the number of sensor nodes and the extensive human efforts for protocol conformance testing in large-scale networks.

Algorithm 1 shows how to virtualize different topologies when testing various routing protocols. In this case, the routing protocol is IUT, and the real sensor node installed with IUT is the SUT. The key point to test the SUT in any given topology is to analyze the effects of different packet transmissions in the SUT, and then to virtualize the effects for the SUT with the tester.

Algorithm 1 Topology virtualization process for any routing protocol.

1: Understand the route discovery mechanism
2: Divide the routing positions in a route into $source$, $forwarder$, and $destination$
3: Divide the nodes into two sets: N_1 (SUT's 1-hop neighbors) and N_2 (SUT's non-neighboring nodes)
4: **while** P takes each value in the above three positions **do**
5: Analyze the effects on the SUT when nodes $\in N_1$ send a packet
6: Analyze the effects on the SUT when nodes $\in N_2$ send a packet
7: Analyze the responses of nodes $\in N_1$ when the SUT sends a packet
8: Analyze the effects of the above responses on the SUT
9: **end while**
10: Build a model M for effects of packet transmission on the SUT from the previous analysis
11: Let S be the set of to-be-tested topologies
12: **while** Based on the model M, $\forall Topo \in S$ **do**
13: Test the SUT when it is communicating with nodes $\in N_1$
14: Test the SUT when it is communicating with nodes $\in N_2$
15: **end while**

B. Event virtualization

Software bugs are often detected when there are non-deterministic events in the network, such as node reboot and outage, packet duplication, packet loss, and so on. These events have potential to drive a WSN application into corner-case situation, where bugs usually lurk. In order to ensure high coverage testing, RealProct employs event virtualization to facilitate protocol conformance testing with different kinds of events, including the non-deterministic ones that are caused by the volatile environment. During the simulations, such events are easy to be inserted. While in real deployments, such events are only allowed to be observed. However, in the testing of RealProct, these events need to be triggered with different methods. The key problem is how to trigger these events with only a few real sensor nodes so as to test the corner-cases. We discuss the virtualization of events below.

1) Node reboot and outage: Node reboot cases are classified into two categories: the SUT reboots and the tester reboots. In the first category, our test cases target at verifying that the SUT can successfully reboot and return

to the initial state. For this case, we just send to SUT the reboot command '*/home/user/contiki-2.4/tools/sky/msp430-bsl-linux –telosb -c /dev/ttyUSB0 -r*' in our experiments. For the second category, the test cases are checking whether the SUT can appropriately handle the reboot of other sensor nodes in various topologies. We do not care about which node reboots in what kind of virtualized topologies. As long as we know the influence of its reboot in the SUT, we can virtualize node reboot for the SUT by exerting such impacts on it.

We take the rime mesh routing protocol (RMRP) used in Contiki with the virtualized topology shown in Fig. 3 as an example. The SUT is the only 1-hop node from the sink, and the application running in the SUT and the tester is sending data to the sink regularly. The reboot of node 1 has small effects on the SUT. Before reboot, node 1's influence on the SUT is its regular data packets and the related route discovery packets. Hence we virtualize its reboot by not allowing the tester to send data packets and route discovery packets with source as node 1 to the SUT. Given that node outage will cause node reboot, the virtualization method is the same.

2) Packet disorder, duplication, corruption, and loss: Existing tools, such as KleeNet, have not explored packet disorder in testing. It is because these tools rely on simulation, model based on human experience [6]. Neither have we realized it until we observed this phenomenon in our testing with real sensor nodes. We reported this problem to the Contiki developers and believed that it was caused by the implementations of the lower level protocols and the volatile wireless channels.

Let us look at the testing of a TCP implementation as an example. Consider the whole topology in Fig. 3, the SUT is the TCP server and node c is the TCP client. From node c to the SUT, there exist two routes. One is c $\rightarrow$ b $\rightarrow$ 2 $\rightarrow$ SUT, and the other is c $\rightarrow$ 3 $\rightarrow$ 1 $\rightarrow$ SUT. If the routing protocol uses two routes simultaneously, then the packets arrived at the SUT from node c may be disordered. Say, the first data packet from node c takes route 1 and the second data packet follows route 2. If route 1 is more congested than route 2, the second data packet may arrive at the SUT before the first data packet, resulting packet disorder at the SUT. If the SUT cannot handle the disorder appropriately, it fails the test case. To virtualize the packet disorder with only one tester, it prepares two packets: packet 1 with a smaller sequence number and packet 2 with a larger sequence number. Then it sends packet 2 to the SUT first, and packet 1 next.

Packet duplication can be virtualized by sending the same packets to the SUT twice or multiple times with different intervals. Similarly, packet corruption is virtualized by distorting different fields of the packet before sending the packet to the SUT. In order to virtualize the packet loss, the tester will not send the expected packet or acknowledgment to the SUT.

C. Dynamic test execution

In simulations, we can control when and where the non-deterministic events occur. However, testing with real sensor nodes is not entirely under human control, and hence we cannot avoid their occurrences in the cases that we do not anticipate. Here we focus on unexpected packet loss caused by the volatile environment, which is the main problem that obstructs our normal testing with real sensor nodes. The loss may occur to any packets we send or receive during our testing, and hence threaten our testing and could result in false positive (FP) and false negative (FN) errors in the test verdict.

PO3 is thus deployed to monitor the difference of packets exchanged between PCO1 and PO3 during execution (see Fig. 2). It can assist human in debugging problems caused especially by packet loss. Actually, the most convenient way to avoid FP and FN errors is to repeat the test cases several times so as to filter out the volatile influence from the environment. Intuitively, the more times we repeat, the less likely that FP and FN errors will occur. However, it is inefficient to execute the test cases for too many times. Thus, here is the problem: What is the minimal number of executions that can guarantee the FP and FN error rates lower than a certain level? We design an algorithm to tackle this problem.

Let the empirical probability of packet loss caused by the environment be L_0. Due to packet loss, the test results will be distorted with chance L_0, and lead to FP and FN errors. Suppose a test case is executed n times, and it passes n_1 times and fails n_2 times. If $n_1 > n_2$, we would like to declare the test as pass (**Assertion 1**). If $n_2 > n_1$, we would like to declare the test as failure (**Assertion 2**). If $n_1 = n_0$, we execute the test case for more times until $n_1 = n_0$ does not hold. However, Assertion 1 may suffer FN error if the test verdict should be fail, but was wrongly concluded as pass. Assertion 2 may suffer FP error if the test verdict should be pass, but was wrongly concluded as fail. What is the minimum value of n that guarantees both Assertion 1 suffers FN error and Assertion 2 suffers FP error at a rate lower than a threshold E, say 1%?

For a test case with $n_1 > n_2$, if FN error occurs, then n_1 passes are caused by packet loss and n_2 failures are not affected by packet loss. Hence the FN probability $P(FN)$ is

$$\binom{n}{n_1} L_0^{n_1}(1-L_0)^{n_2}.$$

Let $P(FN) \leq E$, then the smallest n exists if $n_1 = n$, i.e., all the executions pass. Hence we have the minimum n as

$$n = \lceil \lg_{L_0}^{E} \rceil.$$

Similarly, for a test case with $n_2 > n_1$, the minimum n is also

$$n = \lceil \lg_{L_0}^{E} \rceil.$$

In general, for each test case, it is first executed for $\lceil \lg^{E}_{L_0} \rceil$ times. If the result is $n_1 > n_2$, then we calculate $P(FN)$ and compare it with E. If $P(FN) \leq E$, then we end the test execution and declare the test case as pass. If the result is $n_2 > n_1$, then we calculate $P(FP)$ and compare it with E. If $P(FP) \leq E$, we end the test execution and declare the test case as failure. Otherwise, we repeat the test case execution until $n_1 > n_2$ and $P(FN) \leq E$ or until $n_2 > n_1$ and $P(FP) \leq E$. In this way, the execution repetition of each test case can be minimized in real-time dynamically according to the requirement of the FN and FP error rates.

VI. Generality of RealProct

RealProct presents a generic framework for testing a wide range of protocols in WSNs by using only two real sensor nodes and a PC. Three techniques are designed to make the framework practical for WSNs. These techniques and related algorithms are computed in the PC to relieve the burden of resource-limited sensor nodes.

Although testing diverse protocols in WSNs requires different manual efforts [8], RealProct provides a generic framework and universal techniques to keep the testing process the same and easy to follow. A user only needs to design abstract test cases according to the protocol specification, and then abstract the interfaces of the protocol implementation and translate those abstract test cases into executable ones with our virtualization techniques. Next, the executable test cases will be stored in the PC and downloaded into the tester node in real-time. Our test execution and verdict algorithm can control their execution and verdicts the test results autonomously. Finally, the PC can repeat those failed test cases to help debugging the problems.

VII. Evaluation

This section evaluates the practicability and efficiency of our RealProct for protocol conformance testing with two case studies. We test the TCP protocol of the μIP TCP/IP stack for WSNs in Contiki-2.4 because TCP is not only viable for WSNs, but also very useful to connect WSNs with traditional networks [13]. During the TCP testing, RealProct finds not only two new bugs acknowledged by the developers, but also all the previously discovered bugs [6]. The Rime mesh routing protocol (RMRP) designed for WSNs in Contiki-2.4 is also tested. The test cases for detecting those bugs are available online[1].

A. Detecting new bugs in TCP

Although we design many ordinary test cases based on the TCP specifications, we only detail those test cases that identify bugs.

[1]http://www.cse.cuhk.edu.hk/~jjxiong/testcases/testcase.htm

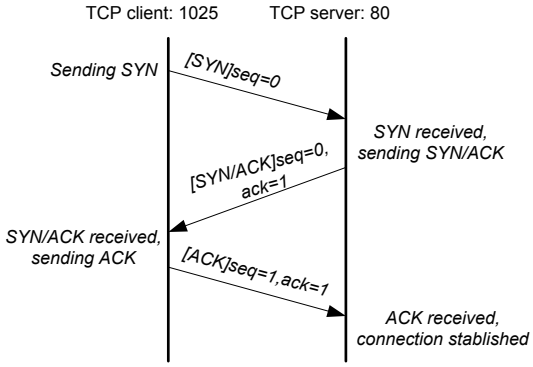

Figure 4. TCP three-way handshake.

1) Bug 1 – Connect to opened TCP ports: We discuss each test case in the aspect of the test purpose, test scenario, test steps, and test results. Since the SUT is not under direct control, we steer it passively by manipulating the tester.

We want to test whether the TCP server allows successful connection to an opened TCP ports. According to Internet Assigned Numbers Authority (IANA), the legal ports for TCP connection are from 0 to 65535. We can test each port one by one because such kinds of similar test case generation and execution are easy to be automated, but it takes long time to execute. As a result, we choose several representative ports to test. We select port 80 as a common port and port 0 as a special port to test. Then we get two test cases.

Test case 1 – Test purpose: Connection to an opened common TCP port should be successful.

Test scenario: TCP is the IUT. One sensor node is the SUT, and it functions as the TCP server installed with the IUT. The UT runs on top of the IUT. It is a simple application which just receives packets and sends the received packets to the peer. The other node is the tester, in which the LT (or tester) is installed with the test case. It functions as the TCP client.

Test steps: The testing steps are as the three-way handshake process of TCP as shown in Fig. 4. First, the server opens port 80 and waits for connection. Second, the tester sends an SYN packet to port 80 of the SUT. Then it waits for the SYN/ACK packet from the SUT. If it does not receive the packet, it declares the failure of the test and exits. Otherwise, it checks the content of the packet. If the content is incorrect, it claims test failure. Otherwise, it replies an ACK packet to the SUT and asserts test case pass.

Our method is different from KleeNet [6] in that only the node to be tested is installed with the IUT. The other node is installed with test cases. Each test case realizes a simple function of the IUT so that it can work as a peer of the IUT and stimulates the IUT in many aspects.

Test results: According to the logs observed at the PCOs and PO, this test case passes. However, **Test case 2** which tests port 0 fails. It seems that although the SUT opens port

0, it does not accept connection to port 0 from TCP client.

Then we repeat the test case and observe the packet logs again. We find that the reason that test case 2 fails is because the client times out waiting for the SYN/ACK packet from the server. That is to say, the server does not reply the SYN to port 0. We also observe "tcp: zero port." in the log. Why does this not happen to port 80? By digging into the code that is related to the ports and the log, we find a segment of code as follows:

```
1 // Make sure that the TCP port number is not zero.
2 if(BUF->destport == 0 || BUF->srcport == 0)
3 {
4     UIP_LOG("tcp: zero port.");
5     goto drop;
6 }
```

This code indicates that if the source port or destination port number is 0, then it just ignores and drops the packet. This explains why the failure happens. Actually, this operation does not conform to the standard TCP specification, and it obstructs the communication between other versions of TCP implementation. By browsing the previous bugs in Contiki-2.x, we find that this bug is introduced after fixing the previous bug. Since RealProct enables high test coverage and rerunning of test cases, it can effectively discover bugs in the latest version of Contiki as shown in this case.

2) Bug 2 – Connect to unopened TCP ports: To complement the previous aspect of port testing in TCP, we should test whether the TCP server allows successful connection to an unopened TCP ports. According to the specification, if the client connects to an unopened TCP ports, the server should reply a reset packet. Similarly, we also test two ports with two test cases.

Test case 3 – Test purpose: Connection to an unopened normal TCP port should be unsuccessful.

Test scenario: It is the same as test case 1.

Test steps: First, the server opens port 80 and waits for connection. Second, the tester sends an SYN packet to port 79 of the SUT. Then it waits for SYN/ACK packet from the SUT. If it does not receive an RST packet from the server, it declares the failure of the test and exits. Otherwise, it checks the content of the packet. If the content is incorrect, it claims test failure. Otherwise, it asserts test case pass.

Test results: pass. However, **Test case 4** in which the client connects to port 0 of the server fails because it gets no RST reply after sending SYN. The reason is the same as the previous bug. A mistake made in the program may result in more than one failure in execution. Since the test execution in RealProct can be easily automated, we would suggest software engineers to rerun the test cases after fixing the bugs.

B. Detecting previous bugs in TCP

Apart from the new bugs, RealProct also detects the four old bugs discovered by KleeNet earlier in the TCP stack implementation of Contiki [6]. The test cases for discovering them are very short and simple. Hence, we only discuss the test cases that identify the bugs caused by non-deterministic events because these bugs will invoke our virtualization techniques.

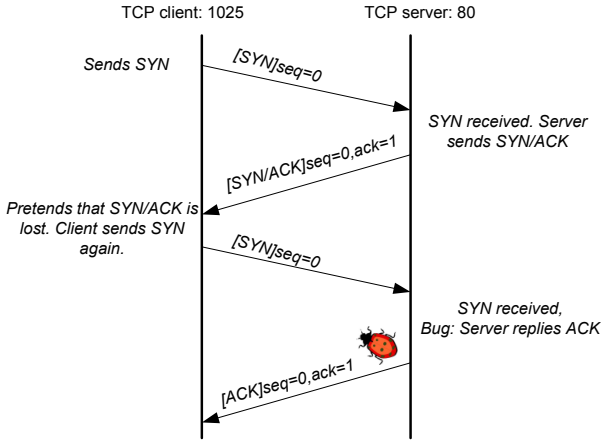

Figure 5. SYN packet loss.

According to the TCP specifications, packet loss leads to retransmission. As long as the loss is not caused by disconnected or extremely poor physical medium, it should not result in TCP connection failure. For example, SYN packet loss would arouse SYN retry for utmost 7 times. If one SYN packet is successfully received before retransmitting the SYN packets 7 times, then the three-way handshake process would continue. Next we discuss the test case that checks the loss of SYN/ACK packet.

Test case 5 – Test purpose: An SYN/ACK packet loss should not result in TCP connection failure.

Test scenario: It is the same as test case 1. The SUT is the TCP server, and the tester is the TCP client.

Test steps: As shown in Fig. 5, first, the server opens port 80 and waits for a connection. Second, the tester sends an SYN packet to port 80 of the SUT. Then it expects an SYN/ACK packet with correct content from the SUT. After receiving and verifying the content of the packet, it ignores the packet and sends an SYN packet to the SUT again as if the SYN/ACK packet is lost. It will declare test pass unless it receives an SYN/ACK packet with correct content again.

Test results: fail. The reason is that the TCP server replies an ACK packet to the TCP client at the second time when it receives an SYN packet. However, the correct behavior for the TCP server is to send an SYN/ACK packet with right content according to the specification. This error leads to two bugs discovered earlier by KleeNet, which are also easily identified by RealProct. As long as this error is fixed, multiple failures can be solved. Compared with KleeNet which has to explore many redundant paths and manually write many assertions, our test cases are clearer, more concise, and more specific. Besides virtualizing packet

loss flexibly, we can also generate extra test cases easily by retransmitting the SYN packets from 2 times up to 6 times on Contiki-2.4 to verify whether this bug is completely fixed. The results show that the above test cases all pass.

Similarly, we also design a test case that virtualizes SYN packet duplication event to uncover a bug. The process is shown online[1].

C. Testing routing protocol RMRP

For RMRP testing, we give two test cases demonstrating the topology virtualization techniques presented in Section V. The first test case checks whether the SUT can establish route to 1-hop neighbor in the topology shown in Fig. 3.

Test case 6 – Test purpose: The SUT can establish route to 1-hop neighbor 2.

Test scenario: RMRP is the IUT. One sensor node is the SUT, and it functions as the route discovery originator installed with the IUT. The UT runs on top of the IUT. It is a simple application that just receives packets and sends the received the packets to the tester. The other node is the tester, in which the LT is installed with the test case. It functions as all the other nodes in the topology.

Test steps: First, the SUT broadcasts a route request packet (REQ) before sending a data packet to node 2. Second, the tester virtualizes node 2 to uni-cast a route reply packet (REP) to the SUT and then virtualizes node a to forward the REQ. Third, tester virtualizes node 2 to wait for the data from the SUT. On receiving the data, it will declare test pass. Otherwise, it will assert failure.

Test results: The pass result validates the RMRP in aspect of route discovery with 1-hop neighbor.

The second test case tests if the SUT can establish route to non-neighboring nodes in the topology shown in Fig. 3. Since this topology will result in four scenarios. The test case is divided into four sub-cases. We also show one sub-case.

Test case 7 – Test purpose: The SUT can establish route to non-neighboring nodes d.

Test scenario: It is the same as test case 6.

Test steps: First, the SUT broadcasts an REQ before sending a data packet to node d. Second, the tester virtualizes node 2 to forward the REQ and then virtualizes node a to forward the REQ. Third, after a while, tester virtualizes node 2 to uni-cast the REP originated from node d to the SUT. Finally, tester virtualizes node d to wait for the data from the SUT. The next-hop of the data should be node 2. If it receives the data, it will declare test pass. Otherwise, it will assert failure.

Test results: The pass result validates the RMRP implementation in aspect of route discovery and initiation with non-neighboring nodes.

[1]http://www.cse.cuhk.edu.hk/~jjxiong/testcases/testcase.htm

VIII. Conclusions

In this paper, we presented RealProct, a reliable protocol conformance testing approach using real sensor nodes to check the conformance of protocol implementation to the specification. RealProct includes two real sensor nodes and a PC for testing and three techniques are carefully designed to support the resource-limited sensor nodes. We proposed topology virtualization as a technique for the two sensor nodes to imitate larger WSNs with different topologies, which allows efficient and flexible protocol conformance testing. Event virtualization is also invented to generate non-deterministic events to support virtualized topology with multi-hop routing. We further enhanced the efficiency of testing and the accuracy of verdict by determining the optimal number of test executions, while guaranteeing an acceptable probability of false negative and positive errors. We implemented RealProct in real sensor nodes and tested the protocol implementation of the μIP TCP/IP stack in Contiki-2.4. RealProct effectively found two new bugs and all the previously detected bugs in the μIP TCP/IP stack. The experiments demonstrated that our protocol conformance testing setup can effectively detect bugs that might be missed in simulations. We also tested and validated the protocol implementation of Rime mesh routing in Contiki-2.4. The results indicated that RealProct can provide a cost-effective and flexible solution for testing different advanced protocols, like multi-hop routing, which are more sensitive to the network topology and the non-deterministic events in real wireless channels.

Due to the limited number of real sensor nodes used, there are some cases that RealProct cannot virtualize with high similarity. For example, It cannot virtualize the wireless environment when interference caused by simultaneous transmissions happens, although the result of interference is easy to be virtualized as packet loss or erroneous packet. In the future, we will attempt at solving the problem with more real sensor nodes.

Acknowledgments

The work described in this paper was supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4154/10E) and sponsored in part by the National Basic Research Program of China (973) under Grant No. 2011CB302600 and the VINNMER program funded by VINNOVA, Sweden.

References

[1] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli, "The hitchhiker's guide to successful wireless sensor network deployments," in *Proc. of the 6th International Conference on Embedded Networked Sensor Systems (SenSys)*, 2008, pp. 43–56.

[2] G. W. Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, USA, 2006, pp. 381–369.

[3] K. Langendoen, A. Baggio, and O. Visser, "Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture," in *Proc. of the International Workshop on Parallel and Distributed Real-Time Systems*, April 2006.

[4] Y. Zhou, X. Chen, M. R. Lyu, and J. Liu, "Sentomist: Unveiling transient sensor network bugs via symptom mining," in *Proc. of the International Conference on Distributed Computing System (ICDCS)*, Genova, Italy, June 2010, pp. 784–794.

[5] P. Li and J. Regehr, "T-Check: Bug finding for sensor networks," in *Proc. of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2010, pp. 174–185.

[6] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weisez, S. Kowalewskiz, and K. Wehrle, "KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment," in *Proc. of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2010, pp. 186–196.

[7] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong, "Passive diagnosis for wireless sensor networks," in *Proc. of the 6th International Conference on Embedded Networked Sensor Systems (SenSys)*, 2008, pp. 1132–1144.

[8] *Information Technology, Open Systems Interconnect, Conformance Testing Methodology and Framework.*, ISO. International Standard IS-9646, 1991.

[9] *Agilent 6430 WiMAX Protocol Conformance Test (PCT) and Development System*, Agilent, http://cp.literature.agilent.com/litweb/pdf/5989-7513EN.pdf.

[10] *Protocol Conformance Testing of 3G Terminals*, Anritsu, http://www.eu.anritsu.com/news/default.php?id=718.

[11] J. Neyman and E. S. Pearson, "On the problem of the most efficient tests of statistical hypotheses," *Royal Society of London Philosophical Transactions Series A*, vol. 231, pp. 289–337, 1933.

[12] A. Dunkels, T. Voigt, J. Alonso, H. Ritter, and J. Schiller, "Connecting wireless sensornets with TCP/IP networks," in *Proc. of the 2nd International Conference on Wired/Wireless Internet Communications (WWIC)*, 2004, pp. 583–594.

[13] A. Dunkels, T. Voigt, and J. Alonso, "Making TCP/IP viable for wireless sensor networks," in *Proc. of the 1st European Workshop on Wireless Sensor Networks (EWSN),*, Berlin, Germany, 2004.

[14] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proc. of the 29th Annual IEEE International Conference on Local Computer Networks (LCN)*, 2004, pp. 455–462.

[15] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, pp. 4–12, 2001.

[16] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, "Efficient memory safety for TinyOS," in *Proc. of the 5th International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007, pp. 266–279.

[17] N. Kothari, T. Millstein, and R. Govindan, "Deriving state machines from TinyOS programs using symbolic execution," in *Proc. of the 7th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2008, pp. 271–282.

[18] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An operating system for sensor networks," in *Ambient Intelligence*. Springer Verlag, 2004, pp. 115–148.

[19] Network simulator NS-2. http://www.isi.edu/nsnam/ns/.

[20] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire tinyos applications," in *Proc. of the 1st International Conference on Embedded Networked Sensor Systems (SenSys)*, 2003, pp. 266–279.

[21] W. P. Timing and B. L. Titzer, "Avrora: Scalable sensor network simulation," in *Proc. of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2005, pp. 477–482.

[22] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[23] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the sensor network debugger," in *Proc. of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys)*, 2005, pp. 255–267.

[24] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse, "Clairvoyant: A comprehensive source-level debugger for wireless sensor networks," in *Proc. of the 5th International Conference on Embedded Networked Sensor Systems (SenSys)*, 2007, pp. 189–203.

[25] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han, "Dustminer: Troubleshooting interactive complexity bugs in sensor networks," in *Proc. of the 6th International Conference on Embedded Networked Sensor Systems (SenSys)*, 2008, pp. 99–112.

[26] Q. Cao, T. F. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks," in *Proc. of the 6th International Conference on Embedded Networked Sensor Systems (SenSys)*, 2008, pp. 85–98.

[27] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets," in *Proc. of ACM Conference on Computer Communication (SIGCOMM)*, 2005, pp. 265–276.