Demo: Exploring Concurrent Transmissions with RssiSpy and TrafficBench

Carsten Herrmann
Networked Embedded Systems Lab
TU Dresden, Germany
carsten.herrmann@tu-dresden.de

Marco Zimmerling
Networked Embedded Systems Lab
University of Freiburg, Germany
zimmerling@cs.uni-freiburg.de

Abstract

This demo presents TrafficBench, a new tool suite for the experimental exploration of concurrent transmissions in low-power wireless networks. TrafficBench integrates RssiSpy—a software module that enables continuous RSSI measurements with bit-level time resolution on standard nRF52840 devices—with a scheduling framework that provides a specialized, easy-to-use language to implement communication test patterns. The latter offloads the user from demanding implementation tasks like time-critical interrupt service routines, network bootstrapping, synchronization, and efficient data logging. Thus, TrafficBench greatly simplifies experimentation and allows the user to focus on the interesting questions.

1 Introduction

In the last decade, many communication protocols for low-power wireless mesh networks have been proposed that make use of *concurrent transmissions (CT)* to improve, e.g., dependability, latency, and robustness to topology dynamics [4]. While the benefits of exploiting CT are clearly visible at the application level, their foundation—the physical-layer effects caused by CT—are still not fully understood. A comprehensive, precise, and tractable analytical model that enables reliable end-to-end performance predictions seems not in reach today. To make progress, the variety of impact factors (modulation scheme, radio hardware architecture, wireless channel) necessitates experiments that enable *deep inspection on actual* (i.e., commercial off-the-shelf) *target devices*.

In this demo, we present TrafficBench,[1] a new tool suite that greatly simplifies such (and other) experiments on nRF52840 devices running in Bluetooth Low Energy (BLE) mode by integrating the following components:

(1) a communication scheduling framework that enables the user to formulate test traffic (i.e., timed transmit and receive operations of all nodes) in an easy-to-use schedule description language, which offloads all demanding and error-prone implementation tasks from the user

(2) RssiSpy [3], a new software module that enables deep inspection by adding high-rate (1 Msps) *receive signal strength indicator (RSSI)* sampling to packet reception

(3) optimized post-processing and logging routines to efficiently handle the high data volume emitted by RssiSpy

(4) a skeleton program that performs basic tasks like network bootstrapping (synchronize schedule execution at all nodes), clock synchronization, and packet logging fully automatically in the background.

These components, which are integrated into a ready-to-use device firmware, are complemented by Python scripts that

(5) extract packet records from log files

(6) analyze recorded packets and add derived information (probable transmitter of each received packet, SNR / SINR estimations, user-configurable markers, etc.)

(7) enable direct evaluation and visualization with Glue [1], a GUI program for linked-data exploration.

2 TrafficBench Highlights

There is not enough space to go through all parts here,[2] so we briefly summarize only two key components: the communication scheduling framework and the RSSI data compression stage. RssiSpy is discussed in [3]. While its integration into TrafficBench enables an unprecedented level of detail, it entails the challenge of handling large data sets resulting from the high sampling rate.

Communication scheduling. A toy example schedule is shown in Fig. 1. Schedules are formulated as sequential programs. Currently, the instruction set contains three possible operations: `SLEEP`, `BRANCH`, and `TRX` (transmit/receive). `TRX` instructions are written as blocks, where the body of each block defines the corresponding packet's content.

During the firmware build process, the schedule program is translated to an internal representation and then stored in each node. Behind the scenes, the translation step is performed by the standard GNU assembler, i.e., the schedule from Fig. 1

[1] The source code of TrafficBench is available at `https://gitlab.com/nes-lab/trafficbench`.

[2] All parts can be inspected at our demo.

```
main_loop:

    // transmit a packet from node 1
    TRX_begin               transmitter=1, tx_delay=2000, timeout=1000000
        TRX_data_fixed          0x01, 0x02, 0x03, 0x04  // fixed content
        TRX_data_random         len=4                   // random content
        TRX_data_checkpoint                             // checkpoint
        TRX_data_checkpoint_marker                      // checkpoint marker
    TRX_end

    // synchronous packets from nodes 2 and 3
    TRX_SYNC_begin          transmitter="2,3", tx_delay=2000, timeout=1000000
        TRX_data_fixed          1,2,3,4
        TRX_data_random         len=4
        TRX_data_checkpoint_marker
    TRX_SYNC_end

    // concurrent and individually time-shifted packets from multiple nodes
    TRX_GROUP_begin         timeout=1000000
        TRX_SYNC_begin          transmitter=1, tx_delay=1000, rng_offset=0x01
            TRX_data_random     len=4
            TRX_data_checkpoint_marker
        TRX_SYNC_end
        TRX_SYNC_begin          transmitter=2, tx_delay=1000, rng_offset=0x01
            TRX_data_random     len=4
            TRX_data_checkpoint_marker
        TRX_SYNC_end
        TRX_SYNC_begin          transmitter=3, tx_delay=2000, rng_offset=0x02
            TRX_data_random     len=4
            TRX_data_checkpoint_marker
        TRX_SYNC_end
    TRX_GROUP_end

    SLEEP       5000000         // sleep 5s
    BRANCH      main_loop       // next loop iteration
```

Figure 1: Exemplary communication schedule.

indeed is an assembly program that emits machine code for an internal *virtual machine (VM)*. The schedule instructions represent machine instructions of this VM. They are realized as macros that hide all arduous details from the user. The benefit of this concept is twofold: (i) There is no need to write a specialized parser. (ii) It is possible to use other existing language features like labels, macros, and constants.

After power-up, a user-defined root node starts executing the schedule and transmitting packets when requested. Other nodes enter continuous listen mode until they receive a packet that includes a *checkpoint*. The latter contains state information (e.g., timestamps, VM program counter) that allows the receiver to synchronize its clock and its VM and, consequently, to join schedule execution. So, over time all nodes wake up, and finally all nodes have synchronized clocks and VMs, i.e., they perform the same operations at the same time.

RSSI data post-processing and compression. We use 128 kB of nRF52840's RAM as RSSI data space. It is organized as a ring of rings, and each inner ring belongs to one `TRX` operation. After a packet reception, a fine-tuned post-processing thread cuts superfluous samples, unrolls the inner ring to a linear buffer, and defragments the outer ring. This frees unused space as fast as possible to maximize RAM availability.

Later on, the recorded data is output using a UART connection. Although we optimize the UART routines to parallelize peripheral and CPU activity as much as possible, the UART link is a bottleneck due to its limited baud rate. To mitigate this problem, we add a data compression module between the post-processing and logging stages. This module exploits typical properties of RSSI data streams to implement a very efficient compressor. Its output is binary compatible with deflate [2], so standard tools can be used for decompression.

The key ideas behind this module are as follows. Deflate combines duplicate substring elimination and Huffman coding [2]. Accordingly, the expensive operations are *string search* and *code construction*. Considering RSSI streams, we

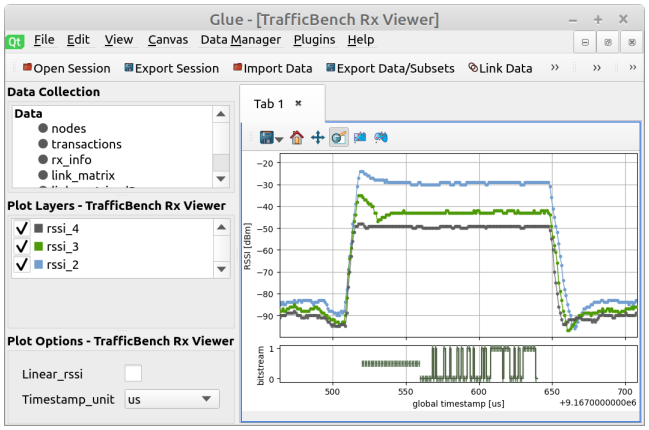

Figure 2: Evaluating recorded data with Glue.

recognize that they are often slowly varying, i.e., except for ramp-up/down phases they are relatively stable or change with limited slope (Fig. 2). Hence, if we encode a sample stream using differences instead of absolute values, the majority of entries is 0 or ± 1. Since this is known a priori, we can construct efficient codes offline and implement the coding stage without dynamic code construction. The string search is significantly accelerated by focusing on strings whose first three bytes stem from $\{-1, 0, +1\}$, which is the majority of strings. Doing so reduces the number of possible beginnings from 2^{24} to 27 and allows bookkeeping on found start sequences, which greatly improves search speed.

3 Demonstration

To showcase the exploration of CT with TrafficBench, we set up a demo with four off-the-shelf nRF52840 DK boards connected to a laptop via USB. The boards form a small wireless network. The USB connections are used for power supply, programming, and logging. Each node runs our firmware with a simple schedule similar to the one shown in Fig. 1. The laptop records each node's UART output and uses our scripts to analyze and visualize the received packets together with the RSSI data streams in Glue (Fig. 2). A single test run takes about one minute, so the interested visitor can have a look at multiple runs and play around, e.g., by changing the nodes' positions or trying different schedules. The demo also illustrates TrafficBench's utility as a general-purpose traffic generator and hopefully provokes interesting discussions regarding potential extensions.

4 Acknowledgments

This work was supported by the German Research Foundation (DFG) within the Emmy Noether project NextIoT (grant ZI 1635/2-1).

5 References

[1] Glue: multi-dimensional linked-data exploration. https://glueviz.org.

[2] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, May 1996.

[3] C. Herrmann and M. Zimmerling. RSSISpy: Inspecting Concurrent Transmissions in the Wild. In *Proc. Int. Conf. Embedded Wireless Systems and Networks*, EWSN '22, Linz, Austria, Oct. 2022.

[4] M. Zimmerling, L. Mottola, and S. Santini. Synchronous Transmissions in Low-Power Wireless: A Survey of Communication Protocols and Network Services. *ACM Computing Surveys*, 53(6):1–39, Nov. 2021.