sciendo CzOTO 2019, volume 1, issue 1, pp. 574-582

ON RAPID DEVELOPMENT OF REACTIVE WIRELESS SENSOR SYSTEMS

doi: 10.2478/czoto-2019-0073
Date of submission of the article to the Editor: 04/11/2018
Date of acceptance of the article by the Editor: 22/12/2018

Paweł Gburzyński[1] – *orcid id: 000-0002-1844-6110*
Elżbieta Kopciuszewska[1] – *orcid id: 0000-0002-4216-6215*
[1]Vistula University, **Poland,** *p.gburzynski@vistula.edu.pl*

Abstract: We present a software platform for designing and testing wireless networks of sensors and actuators (WSNs). The platform consists of three components: an operating system for small-footprint microcontrollers (dubbed PicOS), a software development kit (SDK) amounting to a C-based, event-oriented (reactive) programming language, and a virtual execution platform (VUE^2) capable of emulating complete deployment environments for WSNs and thus facilitating their rapid development.[1] Its most recent incarnation introduced in the present paper is a component of the WSN lab being currently set up at Vistula in collaboration with Olsonet Communications Corporation.[2] We highlight the platform's most interesting features within the context of a production WSN installed at independent-living facilities.
Keywords: wireless sensor networks, people and asset tracking, monitoring, reactive systems

1. INTRODUCTION

A substantial part of our research culminating in the system discussed in the present paper took part in the academe (Akhmetshina et al., 2003; Haque et al., 2009; Boers et al., 2010; Boers et al., 2012); however, our efforts were primary oriented towards the industry within the framework of specific commercial projects, e.g., (Gburzynski et al., 2016), with the objective to effectively, reliably, and quickly build custom WSN applications.
By a WSN we mean a distributed cohort of wirelessly connected nodes processing information acquired from sensors and forwarding that information, or its collectively preprocessed derivatives, to one or more data collection points. The definition covers the obvious class of applications where simple sensor readings, e.g., periodic

[1] Note that the superscript in VUE^2 is part of the name. The abbreviation stands for "Virtual Underlay Execution Engine," i.e., the "E" is "squared."
[2] http://www.olsonet.com

readouts, are passed to a central computer for logging and/or presentation, as well as unforeseen in advance, distributed sensing systems where the network processes the sensor data in a way that exploits the distributed and possibly mobile nature of its nodes. For example, the traffic patterns in our WSN need not be constrained to the unidirectional sensor-to-sink flow, naturally allowing for actuators. Also, some sensors can be *virtual*, i.e., their "readings" may arise from some internal and distributed processing (see Section 4). What we mean is not a simple re-interpretation of a sensor value (like unit conversion or visualization), but some "value added" transformation where the network is an essential enabler.

The primary, generic class of applications envisioned for our networks is *monitoring* appertaining to people, assets, or the environment, e.g., for security or generally interpreted well-being. In contrast to some monitoring systems, e.g., ones involving surveillance cameras, the amount of data carried by our networks is small. A typical sensor reading involves a few bytes. A typical readout frequency of a sensor (translating into the ballpark report frequency of a node) is of order 1Hz (or less).

Having started in a competitive industrial environment, we strived from the very beginning to base our networks on the cheapest (smallest-footprint) hardware available. The nodes of our networks are built around microcontrollers with 1-10 KB of RAM and 20-48 KB of flash ROM outfitted with RF modules operating in the ISM (sub-1GB) RF band at the nominal transmission rate between 4.5 and 200 kbps. They do not adhere to the popular standards, like Bluetooth or IEEE 805.15-4, for internal RF communication, although they can be easily interfaced to sensing and peripheral equipment compatible with those standards.

1.1. Network structure and terminology

In its general view, a WSN constitutes a mesh where all nodes are peers from the viewpoint of communication. The ad-hoc forwarding scheme, known as TARP (Gburzyński et al., 2007; Olesiński et al., 2003), automatically (or semi-automatically)[3] adapts to all reasonable communication patterns required by the application. One can often see two types of nodes: *Tags* implementing the (possibly mobile) leaf devices, typically equipped with sensors, and *Pegs* acting as (mostly sensor-less) access points for the Tags. Such a model, dubbed Tags & Pegs (T&P), would naturally apply to a network deployed within a facility (a building or a campus) with the Pegs acting as a semi-infrastructure. One can also think of a spontaneous network deployed entirely on demand. For example, a (supervised) group of people traveling together, or a set of related goods transported together, might form a network to keep track of the group's well-being. A network built according to this model would be comprised exclusively of Tags assuming both roles, i.e., end devices as well as routers; hence the model's name: Routing Tags (RT). In our approach, models like that become so-called *application blueprints*, i.e., actual, albeit virtual, networks with open-ended functionality and smart parametrization. The idea is to abstract from many application-level details, like the assortment of physical sensors and the functions of the external programs, while fully implementing the network framework. In Section 3, we explain how it is possible to *fully* implement a sensor network without completely specifying its hardware.

[3] In a way controlled by parameters than can be set dynamically by the network manager.

A complete (target) application of a WSN is separated from its blueprint by one more intermediate entity called the *praxis* by which we mean the full collection of programs implementing the *logical* functionality of a given sensor system. In addition to the blueprint WSN, the praxis includes external software interfacing the network to the users (which may be software agents). That software is referred to as the *operational support system* (OSS). The identification of the praxis as an instance of the blueprint makes it possible, e.g., to develop and test the OSS with the real-life, physical network being yet unavailable.

1.2. The challenges

Targeting low-end devices for the network nodes, we wanted to be able to program them comfortably and efficiently, to harness their minuscule resources in the best possible way. Contrary to the popular opinion, especially in the academe, the demand for tiny-footprint microcontrollers is not subsiding, and is not likely to be eliminated soon by the decreasing prices of the larger ones. For illustration, the success of one of our commercial projects hinged on being able to use a microcontroller equipped with 2 KB of RAM instead of its marginally more expensive 4 KB variant. A small-footprint device capable of achieving the same feat as a large one will always win, not only by being cheaper, but also in terms of energy demands and reliability. The problem of maximizing the yield from a tiny microcontroller equipped with an RF module should be viewed holistically because of the multiple facets of the tradeoffs. For example, minimizing the energy usage of a microcontroller (critical from the viewpoint of a battery-powered wireless sensor node) requires a careful approach to programming where the duty cycling utilizing deep-sleep states of the CPU is achieved by a structured collection of multiple reactive threads. One problem is thus fitting those threads into the limited RAM without sacrificing their flexibility. For that we needed a programming environment that would allow us to apply high-level techniques to produce flexible software with a tiny footprint.
The concepts of blueprints and praxes, coming before applications, is critical from the viewpoint of rapid, effective, and reliable development. A blueprint and a praxis can be procured and tested in advance with the intention to be used as the engines of multiple *actual* application. Having an environment where this kind of development process can be carried out virtually and authoritatively, with the deployment of physical devices only happening as its final, crowning step, is a paramount advantage, not only in industrial projects but also in education.

2. THE PROGRAMMING ENVIRONMENT

The most serious problem with multithreaded programming in a tiny-RAM environment is the stack space which tends to fragment the scarce memory. With the traditional approach to multithreading, every thread needs a private and comfortably-sized chunk of stack to be able to execute functions and preserve context. One popular approach in tackling this issue, e.g., adopted in TinyOS (Levis & others, 2005), is to drastically limit the number of threads (e.g., to one) delegating the concurrency to the interrupt service functions. This is restrictive from the programmer’s viewpoint as well as prone to reliability problems (Regehr et al., 2005).

```
fsm hrate {
  byte HeartRate;
  state HR_INIT:
    delay (3 * 1024, HR_SEND);
    release;
  state HR_SEND:
    byte cr;
    if ((cr = hrc_get ()) != 255)
      HeartRate = cr;
    if (XWS == 0) {
           address packet;
      packet = tcv_wnp (HR_SEND, BSFD, 2);
      put1 (packet, PT_HRATE);
      put1 (packet, HeartRate);
      tcv_endp (packet);
    }
    proceed HR_INIT;
}
```

Fig. 1. A sample thread in PicOS

2.1. The programming languge

The programming language of our platform has been designed and implemented as an extension of C and integrated with the operating system. As the two are inseparable, it is common for them to be referred to by the same name: PicOS.

Fig. 1 shows a sample PicOS thread. The opening keyword *fsm* stands for "finite state machine." A thread typically consists of a number of *states* interpreted as entry points. A thread declares its willingness to respond to an event by indicating the state to be assumed upon the event's occurrence. For example, by invoking *delay* in its first state (*HR_INIT*), the thread in Fig. 1 indicates that it wants to be run in state *HR_SEND* after the specified delay.

Any reason why a thread may not be able to run (which in a reactive system is the default state of affairs) translates into waiting for a transition to some specific state. A single thread may be waiting for several events at the same time: the first one to occur will trigger the respective transition. This mechanism also applies to thread blocking on I/O or (temporary) unavailability of resources. For illustration, in its second state, the thread attempts to send out a packet, which action consists of acquiring a packet buffer for output (operation *tcv_wnp*), filling it with data, and finally releasing the buffer (with *tcv_endp*). The first step may block on the lack of buffer space. In such a case, the function will not return, and the thread will end up waiting for an event, to be resumed in the same state *HR_SEND* when the buffer space becomes available.

While waiting for an event, to enter or re-enter one of its states, a thread uses no stack space. The only context information needed to continue its execution is the state identifier. This approach strikes a compromise between the preemption opportunities (a thread can only lose the CPU at a boundary between states) and the amount of RAM needed for multithreading. Any "local" variables declared by the thread after the opening keyword, e.g., *HeartRate* in Fig. 1, are in fact *static*, i.e., they are not allocated on the stack (say, as in plain C). A truly local (automatic, in C parlance) variable, exemplified by *cr* in Fig. 1, must be declared within the scope of one state, and it does not survive state transitions. This way, all threads can share the same stack which is also safely used by the interrupt service functions of the

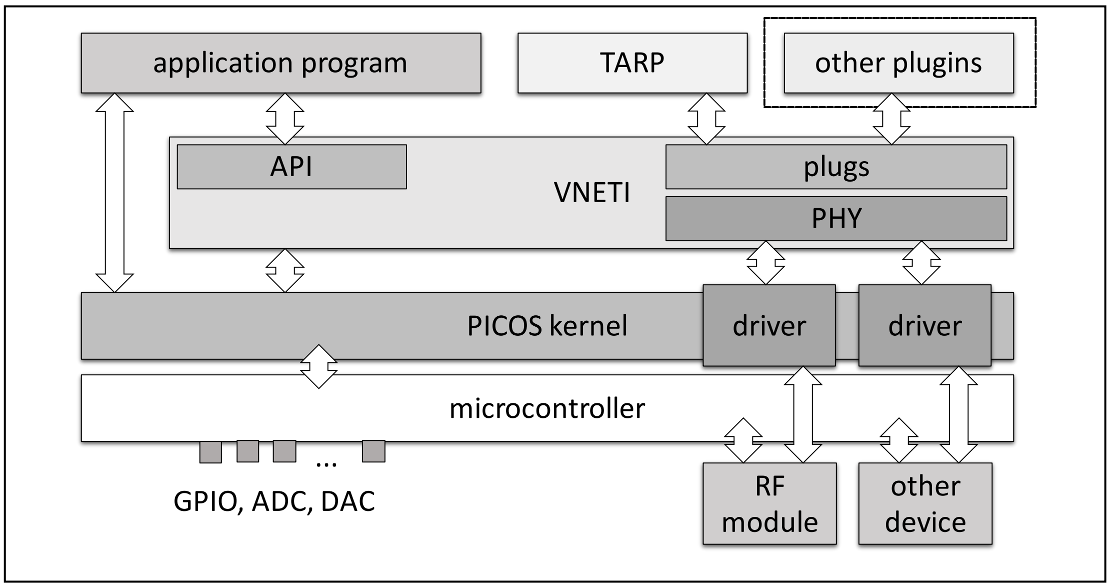

Fig. 2. The operating system layout

operating system. This compromise appears to nicely fit the reactive paradigm of threads running in a wireless mote.

2.2. The operating system

The OS layout is shown in Fig. 2. The system features a holistic I/O API (dubbed VNETI) where drivers are implemented as plugins supplementing a blanket default functionality. New plugins can affect (modify, complement) other plugins, which makes it easy to safely and structurally re-parameterize blueprints and praxes to slightly different requirements and expectations of specific applications.

3. VIRTUAL EXECUTION

The FSM-based programming paradigm was inspired by SMURPH (Dobosiewicz et al., 1993) (Gburzyński and Nikolaidis, 2006) which is a specification system and low-

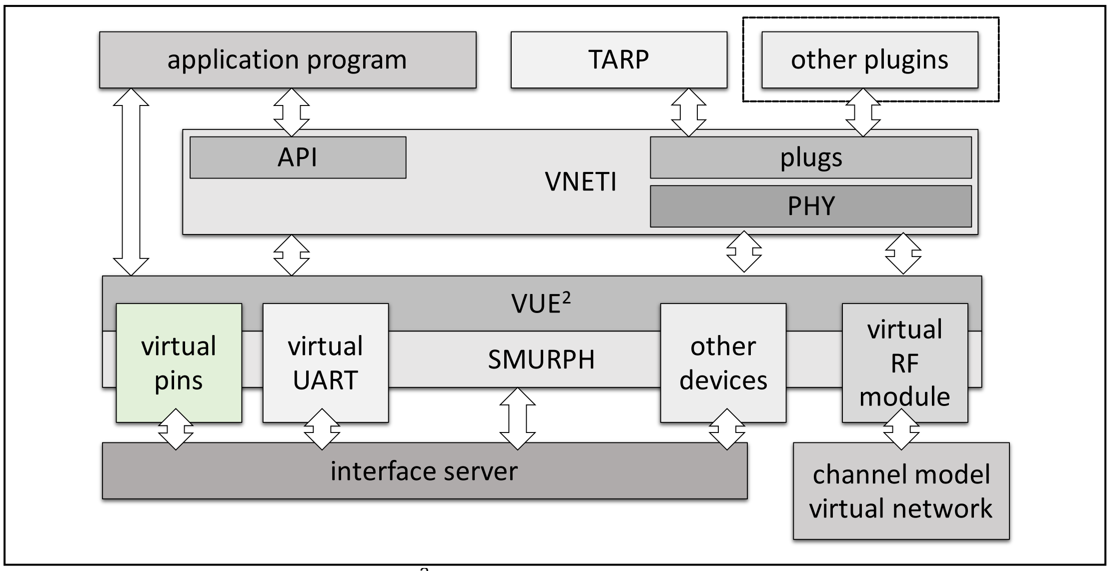

Fig. 3. VUE^2 emulation of a PicOS application

level simulator for communication networks and protocols. The closeness of the two programming environments made it possible to think of mechanically translating

PicOS programs into SMURPH code whereby the resulting SMURPH model describes a complete network whose detailed configuration is parameterized by additional data. This is shown schematically in Fig. 3 which, in confrontation with Fig. 2, identifies the demarcation point where the SMURPH model takes over from PicOS. In contrast to the popular approach of emulation at the microcontroller level, we take a bypass into the SMURPH model by recompiling the network part of the praxis into code acceptable by SMURPH. This is handled by the same PicOS compiler which, in its original guise, transforms PicOS programs into C. One advantage of this approach is its independence of the microcontroller. Another advantage is the immediate access of the model to the plethora of SMURPH tools for describing networks and (wireless) communication channels. In particular, the step where the set of programs intended for the different types of physical nodes are transformed into instances executed at multiple copies of their virtual counterparts is handled elegantly by encapsulating C code into C++ objects owned by instances of node classes.

One can see now how the virtual execution in VUE^2 naturally facilitates the praxis concept. In the model, it is quite natural to abstract from the physical idiosyncrasies of specific sensors and actuators, substituting for them some abstract objects generating the requisite data, e.g., handled directly by the developer (from a GUI) or by external agents that can be attached to the model via the interface server (Fig. 3). Most notably, the OSS programs for the complete application can be fully developed by interacting with the model. With an authoritative virtual replica of the network in the development loop, the OSS programs can be developed faster and tested much better than with the real network, by being easily exposed to extreme and abnormal conditions, which may be difficult to come by in the real (production) world. This is not entirely unlike using an aircraft simulator for testing exceptional (or physically dangerous) scenarios.

4. A SAMPLE APPLICATION

For a real-life implementation of the T&P blueprint, consider the network, dubbed Alphanet in the sequel, deployed at a number of independent living (IL) facilities in Belgium and France (Gburzynski et al., 2016). The network's goal is to detect *events* (or *alarms*) and communicate them to the OSS. Typically, those events are indicative of anomalies requiring personnel's attention. The Pegs jointly form the network fabric whereby alarms reported by the Tags propagate, possibly over multiple hops, towards the so-called *master* Peg directly connected to the OSS. Any Peg can pick up and forward any packet addressed to the master, be it an alarm packet directly issued by a (nearby) Tag or a report relayed by a nearby Peg. TARP (Gburzyński et al., 2007) dynamically and automatically adjusts the path redundancy in the mesh of Pegs to make sure that the events are delivered to the master with satisfying reliability. The praxis assumes that Tags never forward packets. Most of them are battery-powered, so their energy budget is critical. Pegs, on the other hand, are deployed at fixed (inconspicuous) locations and powered from outlets. Both node types are built around the same CC430F6137 microcontroller by Texas Instruments equipped with an ISM RF module (Texas Instruments, 2013). The nominal data rate of the wireless channel is 38,400 bps.

In addition to obvious event triggers, like physical sensors and panic buttons, the system implements several types of virtual sensors whose role is to evaluate distributed alarm predicates, e.g., resulting from the (sensed) proximity of Tags to other devices. The most interesting of them is the “location sensor,” i.e., a full lightweight location tracking system implemented as a relatively simple add-on to the network’s basic functionality. The tracking problem is defined as identifying the room (or area) when the Tag triggering a particular class of alarms is located, so the personnel can promptly locate the patient in need. The sole physical prerequisite is the RSS (received signal strength) measurement from packets received by the Pegs from the tracked Tag. The surprisingly high accuracy of the Alphanet location tracker, which from the perspective of heavy-weight systems (Adame et al., 2018) is based on trivial and cheap assumptions, results from a creative application of multiple power levels to a series of short packet bursts emitted by the Tag along with the alarm event. The creativity (and the resulting quality) is in the interpretation of the readings incurred by those bursts at the neighboring Pegs.

5. SUMMARY

The most interesting feature of our platform is its unique ability to execute complete WSN applications in a virtual environment. The execution is authoritative, i.e., the applications can be virtually deployed and tested without the need to install physical devices. This shortens the development cycle and makes the product more reliable by exposing it to possibly stressful and exhaustive tests that may be difficult (or impossible) to carry out in a production system. Note that testing and debugging a physically deployed network may be a tedious process calling for frequent code replacement in multiple devices possibly distributed over a large area, often in difficult to access places. Several tools have been proposed to assist debugging and code replacement in wireless nodes (Whitehouse et al., 2006), but they all pose tall demands on node resources, be it RAM or the energy budget for RF communication, which makes them inapplicable in our domain. Even with perfect tools, the comfort of armchair development of complete network applications on a laptop will always trump the limited malleability of the real system.

Ad-hoc communication may not immediately appear as strikingly advantageous in an environment where solid infrastructure (Ethernet, WiFi) is a norm and the routing nodes are effectively nailed to the wall (Section 4). Note, however, that the ad-hoc communication paradigm makes the WSN independent of the infrastructure and thus more reliable, which means more than merely making the system less prone to power failures or disasters (Wang et al., 2016). A more important advantage is in rendering the system portable, so, for example, it can be taken “on the road”' or quickly deployed (in an ad-hoc manner), e.g., for a special (external) event.[4] Most importantly, being able to fully control all the communication nodes contributing to the WSN makes it easier to implement *virtual sensors.* The kind of accurate tracking system implemented in Alphanet could only be realized in a network whose infrastructure was in our full control.

Last, but not least, the platform is useful for education. The WSN lab currently being set up at Vistula will play a two-fold role: enabling practical experiments with real-life wireless sensor networks (both for research and education) and equipping the researcher or student with a cozy environment for building such networks on a laptop

[4] The Pegs can be powered from batteries at acceptable (in such circumstances) energy budget.

at home, thoroughly testing them and debugging before flashing the programs into the physical devices.

REFERENCES

Adame, T., Bel, A., Carreras, A., Melia-Segui, J., Oliver, M. i Pous, R., 2018. *CUIDATS: An RFID--WSN hybrid monitoring system for smart health care environments.* Future Generation Computer Systems, 602-615.

Akhmetshina, E., Gburzyński, P., & Vizeacoumar, F., 2003. *PicOS: A Tiny Operating System for Extremely Small Embedded Platforms.* Proceedings of ESA'03, (pp. 116-122). Las Vegas.

Boers, N. M., Chodos, D., Gburzyński, P., Guirguis, L., Huang, J., Lederer, R., Stroulia, E., 2010. *The smart condo project: services for independent living.* E-Health, assistive technologies and applications for assisted living: challenges and solutions. IGI Global.

Boers, N. M., Gburzyński, P., Nikolaidis, I., Olesiński, W., 2010. *Developing wireless sensor network applications in a virtual environment.* Telecommunication Systems, 45, 165-176. doi:10.1007/s11235-009-9246-x

Boers, N., Nikolaidis, I., Gburzyński, P., & Olesiński, W., 2012. *PICOS & VNETI: Enabling Real Life Layer-less WSN Applications.* Proceedings of Sensornets'12. Rome.

Dobosiewicz, W., Gburzyński, P., 1993. *SMURPH: An Object Oriented Simulator for Communication Networks and Protocols.* Proceedings of MASCOTS'93, Tools Fair Presentation, (strony 351-352).

Gburzyński, P., Nikolaidis, I., 2006. *Wireless Network Simulation Extensions in SMURPH/SIDE.* Proceedings of the 2006 Winter Simulation Conference (WSC'06). Monetery, CA.

Gburzyński, P., Olesiński, W., 2008. *On a practical approach to low-cost ad hoc wireless networking.* Journal of Telecommunications and Information Technology, 2008, 29-42.

Gburzyński, P., Kaminska, B., Olesiński, W., 2007. *A Tiny and Efficient Wireless Ad-hoc Protocol for Low-cost Sensor Networks.* Proceedings of DATE'07, (pp. 1562-1567). Nice.

Gburzynski, P., Olesinski, W., Van Vooren, J., 2016. *A WSN-based, RSS-driven, Real-time Location Tracking System for Independent Living Facilities.* 13-th International Joint Conference on e-Business and Telecommunications. Lisbon.

Haque, I., Nikolaidis, I., Gburzyński, P., 2009. *A Scheme for Indoor Localization through RF Profiling*. ICC'09. Dresden.

Levis, P., others, 2005. *TinyOS: An Operating System for Sensor Networks.* W W. Weber, J. M. Rabaey, E. Aarts (Editors), Ambient Intelligence (pp. 115-148). Springer.

Olesiński, W., Rahman, A., Gburzyński, P., 2003. *TARP: a tiny ad-hoc routing protocol for wireless networks.* Australian Telecommunication, Networks and Applications Conference ({ATNAC}). Melbourne.

Regehr, J., Reid, A., Webb, K., 2005. *Eliminating stack overflow by abstract interpretation.* ACM Transactions on Embedded Computing Systems (TECS), 4, 751-778.

Texas Instruments, 2013. *CC430 User's Guide.* Available at: http://www.ti.com/lit/ug/slau259e/slau259e.pdf.

Wang, C., Xing, L., Vokkarane, V. M., Sun, Y. L., 2016. *Infrastructure communication sensitivity analysis of wireless sensor networks.* Quality and Reliability Engineering International, 32, 581-594.

Whitehouse, K., Tolle, G., Taneja, J., Sharp, C., Kim, S., Jeong, J., Culler, D., 2006. *Marionette: using RPC for interactive development and debugging of wireless embedded networks.* Proceedings of the 5th international conference on Information processing in sensor networks, (pp. 416-423).