

# Developing wireless sensor network applications in a virtual environment

Nicholas M. Boers · Paweł Gburzyński ·  
Ioanis Nikolaidis · Włodek Olesiński

© Springer Science+Business Media, LLC 2010

**Abstract** We describe our “holistic” platform for developing wireless ad hoc sensor networks and focus on its most representative and essential virtualization component: VUE<sup>2</sup> (the Virtual Underlay Emulation Engine). Its role is to provide a vehicle for the authoritative emulation of complete networked applications before physically deploying any wireless nodes. The goal is to be able to verify those applications *exhaustively* before programming the hardware, such that no further (field) tests are necessary. We explain how VUE<sup>2</sup> achieves this goal owing to several facilitating factors, most notably the powerful programming paradigm that our platform adopts. As implied by the holistic nature of the discussed system, our work touches upon operating systems, simulation, network protocols, real-time systems, and programming methodology.

**Keywords** Wireless sensor network · Application development · Virtualization

---

N.M. Boers · P. Gburzyński (✉) · I. Nikolaidis  
Department of Computing Science, University of Alberta,  
Edmonton, Alberta, Canada, T6G 2E8  
e-mail: [pawel@cs.ualberta.ca](mailto:pawel@cs.ualberta.ca)

N.M. Boers  
e-mail: [boers@cs.ualberta.ca](mailto:boers@cs.ualberta.ca)

I. Nikolaidis  
e-mail: [yannis@cs.ualberta.ca](mailto:yannis@cs.ualberta.ca)

W. Olesiński  
Olsonet Communications, 51 Wycliffe Street, Ottawa, Ontario,  
Canada, K2G 5L9  
e-mail: [wlodek@olsonet.com](mailto:wlodek@olsonet.com)

## 1 Introduction

Although simple wireless devices built from low-end components are quite prevalent these days, one seldom hears about serious wireless networks within this framework. Consider, for example, a key-chain car door opener or remote starter. A networking “node” of this kind is typically built around a low-power microcontroller with about 1 KB of RAM that drives a simple radio transceiver. The combined cost of the two components is usually below \$5. While it is not a big challenge to implement within this framework a simple broadcaster of short packets, it is quite another issue to turn this device into a collaborating node of a serious ad hoc wireless system. Apparently, the plethora of popular ad hoc routing schemes proposed and analyzed in literature [1–8] address devices with a somewhat larger resource base. Indeed, many of the commercially available nodes for ad hoc networking (somewhat inappropriately called “motes”) are in fact quite serious computers with megabytes of RAM and rather extravagant resource demands. To make matters worse, some people believe that such devices must be programmed in Java to make serious applications possible [9].

Within this context, efforts to introduce an order and methodology into programming small devices often meet with skepticism and shrugs. A common misconception is that one will not have to wait for long before these devices disappear and become superseded by larger ones capable of supporting “serious” programming platforms. Despite the ever decreasing cost of microcontrollers, we see absolutely no reduction in the demand for the ones at the lowest end of the spectrum. One should notice that Moore’s law can be interpreted two ways. The second, considerably less popular (but no less important) interpretation is that the ever decreasing cost and footprint of the low-end devices enable their

new applications and bring in the potential of narrowing the gap between the publicized promise of ad hoc wireless sensor networking and the actual state of affairs. However, similar to the impossibility of having the three desirable properties of food (cheap, fast, and good tasting) present at the same time, wireless sensor networking has problems being cheap, ad hoc, and useful, all at once.

Our approach to wireless sensor network development is motivated by a simple objective: to build networked sensing programs (applications) that, with a single copy of the source code, can be executed on an emulation platform *as well as* an actual device. The postulated compatibility is at the source code level, i.e., it is OK to compile the program separately for each of the two cases. Within such a platform, large-scale studies of new applications and protocols can be undertaken in a programmer-friendly way: the programs can be developed and debugged in the emulator before the exact same code is compiled and loaded onto the actual hardware. The overall objective is to rapidly create massive wireless ad hoc sensor networks using the smallest and cheapest devices available today. In our networks, such devices are capable of ad hoc routing while offering enough processing power to cater to complex applications involving distributed sensing and monitoring.

At the lowest level, virtualization can be accomplished by simulating in software a particular instruction set, i.e., by means of a bytecode interpreter. This approach has been the basis even for commercial grade products, but within the scope of our paper the primary example is Maté [10]. The main shortcomings of such an approach are (a) the interpreter overhead (in terms of both space and time), (b) a lack of back ends for compilation from familiar high-level languages into the invented bytecode format, and (c) the need to define the interaction with peripheral components, e.g., transceivers, in a manner consistent with the invented instruction set. If the virtual machine is fairly well established, e.g., JVM, then one can claim that at least (b) and (c) have been addressed. However, there seems to exist no successful virtual machine geared to sensor devices. For example, even virtual machines intended for small footprint devices (like Dalvik VM,<sup>1</sup> part of the Google Android platform<sup>2</sup>) are “heavyweight” for sensor nodes. In addition (as it happens with Dalvik VM), not all of the language libraries are implemented, raising questions on the extent that VMs can be ported to small platforms without compromising fidelity.

On the other end of the spectrum, we find virtualization by means of an API provided by the underlying operating system. The API is accessible using familiar high-level languages like C or C++. One example is the POSIX API. A platform that can provide run-time emulation of an API can

be thought of as successfully virtualizing the real system at the level of the API. Unfortunately, commodity OS APIs are very broad, and the abstractions they promote are expensive to implement on a sensor device. For example, the Berkeley sockets API is a powerful, albeit expensive, way to abstract network communication. Even worse, it is not a useful abstraction for small devices that do not care about a TCP/IP stack.

We believe that a viable compromise between the two extremes is to introduce a small footprint OS, to specify the API supported by the OS, and to subsequently offer virtualization at the level of that API. Note that in following this approach, we do not need a new toolchain for code production, since we neither have to invent a new higher level language nor do we need to procure a compiler back end for a new (invented) instruction set.

In this paper, we focus on the interplay of PicOS (our operating system for tiny microcontrolled hardware [11]) and VUE<sup>2</sup> (the Virtual Underlay Emulation Engine for realistically simulating networked applications programmed in PicOS). The exact same application source code can be compiled for both the physical hardware and the simulator, making it trivially easy to switch between the two targets. Within the simulator, setting up various topologies, mobility, and parameters is easy. Owing to the powerful tools available in the simulator for modeling RF channels, it is possible to study the performance of real-life applications under diverse scenarios that would be difficult to set up in field tests with real nodes. As VUE<sup>2</sup> runs the fully functional code of the complete application (called *praxis* in PicOS), any external agents intended to talk to the real network can be developed and authoritatively tested in a purely virtual environment.

## 2 PicOS

The most serious problem that occurs with implementing non-trivial, structured, multitasking programs on microcontrollers with limited RAM is minimizing the amount of memory needed to sustain a thread. The most troublesome component of the thread footprint is its stack, which must be preallocated to every thread in a safe amount sufficient for its maximum possible need.

As an example, TinyOS [12, 13] addresses the issue of limited stack space in a radical manner. Essentially, it defines two types of activities: *event handlers* (corresponding to interrupt service routines and callbacks) and so-called *tasks* (simply chunks of code that cannot be preempted by, and thus cannot dynamically coexist with, other tasks). We see programming within this paradigm as a weakness for describing serious wireless sensor network systems.

PicOS strikes a compromise between the complete lack of threads and overtaxing the tiny amount of RAM with

<sup>1</sup>See <http://www.dalvikvm.com/>.

<sup>2</sup>See <http://code.google.com/android/>.

fragmented stack space. A thread contains a number of *checkpoints* that provide preemption opportunities. In the imposed structured organization of a thread, we try to (a) avoid locking the CPU at a single thread for an extensive period of time and (b) use the checkpoints as a natural and useful element of a thread's specification to enhance its clarity and reduce its structure's complexity. These ideas lie at the heart of PicOS's concept of multiprocessing: its threads are structured like finite state machines (FSMs) and exhibit the dynamics of coroutines [14, 15] with multiple entry points and implicit control transfer.

The value of FSM-like programming abstractions is evident to anyone developing networking protocols, as most protocols tend to be described, or even formalized, as communicating FSMs. In addition, programming using a coroutine paradigm is a fairly well-accepted approach and has survived in modern languages (e.g., in "stackless" Python [16] and more recently Ruby [17]). The only general criticism coroutines receive is that the lack of arbitrary preemption might allow CPU-bound tasks to monopolize the CPU. This is not a fundamental problem with our approach because a CPU-bound task can be broken down into a sequence of states to allow preemption at state transitions. On the other hand, an important element of our view is to use the coroutine paradigm as a means to *discourage* CPU-intensive tasks on nodes. Instead, any tasks of this kind should be either moved to data collectors (i.e., full-scale computers) or delegated to specialized (possibly reconfigurable) hardware.

## 2.1 A historical note

PicOS has been arrived at indirectly as a step in the evolution of our network simulation system SMURPH [18, 19]. Intended for the accurate expression of low-level phenomena (necessary for high-fidelity modeling of MAC-level protocols [20–23]), SMURPH underwent a number of enhancements and, at some point, became more than just a simulator, e.g., it was equipped with instruments for dynamic conformance testing [24]. Its capability for rigorous representation of all the relevant engineering problems occurring in detailed protocol design turned it into a specification system. Although oriented towards modeling networks and their protocols, SMURPH became a *de facto* general purpose specification and simulation package for reactive systems [25, 26].

A natural next step in SMURPH's evolution was its extension into a programming platform for building distributed controllers of physical equipment represented by collections of sensors and actuators. At that point, SMURPH was renamed SIDE (Sensors In a Distributed Environment) and encompassed the old simulator augmented by tools for interfacing its programs to real-life objects [27, 28].

```

thread (sniffer)
  entry (RC_TRY)
    packet = tcv_rnp (RC_TRY, efd);
    length = tcv_left (packet);
  entry (RC_PASS)
    if (buffer->status != US_READY) {
      when (&buffer, RC_PASS);
      delay (1000, RC_LOCKED);
      release;
    }
    ...
  entry (RC_LOCKED)
    ...
  entry (RC_ENP)
    tcv_endp (packet);
    trigger (&packet);
    proceed (RC_TRY);
endthread
    
```

**Fig. 1** Code for a sample PicOS thread

A highly practical project—the development of a low-cost wireless badge—inspired the idea to implement a complete executable environment for microcontrollers based on SMURPH's programming paradigm. After developing a SMURPH model for the badge, the most reliable way of transporting it to the real device was to implement the target microprogram on top of a tiny execution environment mimicking SMURPH's mechanism for multithreading and event handling [11]. Incidentally, that mechanism facilitated a stackless implementation of multithreading. Consequently, the resultant footprint of the complete application was trivially small (< 1 KB of RAM), while the application itself was expressed as a structured and essentially self-documented program strictly conforming to its SMURPH model.

## 2.2 The anatomy of a PicOS thread

Figure 1 shows a sample PicOS thread. In this C code, new keywords and constructs are straightforward macros handled by the standard C preprocessor. The `entry` statements mark the different states of the thread's FSM.

A thread can lose the processor when (a) it explicitly relinquishes control at the boundary of its current state (e.g., `release`) or (b) a function call blocks (e.g., `tcv_rnp` if no new packets are available in the receive queue). In both cases, the CPU returns to the scheduler, which can then allocate it to another thread. Whenever a thread is assigned the CPU, execution continues in that thread's *current state*.

Before executing `release`, a thread typically issues a number of *wait requests* identifying one or more events to resume it in the future (e.g., `when` for Inter-Process Communication [IPC] and `delay` for timed events). The collection of wait requests issued by a thread in each state describes the dynamic options for its transition function from that state.

In state `RC_PASS` (Fig. 1), when the `if` condition holds, the thread issues two wait requests: one with `when` and the other with `delay`. With `when`, the thread declares that it wants to be resumed in state `RC_PASS` upon the occurrence of an event represented by the address of a data object (buffer). Such events can be signaled with `trigger`, as illustrated in state `RC_ENP`. The `delay` operation sets up an alarm clock for the prescribed number of milliseconds (1000). When the alarm clock goes off, it will trigger an event that wakes the process up

A somewhat less obvious case of a wait request is the operation `proceed` (at the end of state `RC_ENP`), which implements an unconditional transition to the indicated state. It can be thought of as a zero millisecond `delay` request (indicating the target state) followed by `release`. Thus, the transition involves releasing the CPU and re-acquiring it again, which gives other threads an opportunity to execute in the meantime.

PicOS makes a distinction between two slightly different types of threads. An alternative declaration of a thread involves the keyword `strand` (instead of `thread` in Fig. 1), which, in addition to the thread's name, expects a data type. A thread (strand) declared this way will use a private data object identified by a pointer and available whenever the strand is run (see Sect. 5). The general idea is that a *strand* is intended to exist in multiple copies, each copy operating on a different instance of some data structure, while a *thread* typically exists in at most one version at any moment during the program's life time.

### 2.3 System organization

The organization of PicOS is shown in Fig. 2. VNETI (Versatile NETwork Interface) acts as a layerless networking module, whereby the equivalents of "protocol stacks" are implemented as plug-ins. The set of operations available to plug-ins involve queue manipulations, cloning packets, inserting special packets, and assigning to packets the so-called *disposition codes* representing various processing stages. Any protocol can be implemented within this paradigm, with TARP (our Tiny Ad hoc Routing Protocol [29, 30]) being the most prominent example. The modus operandi of VNETI is that packets are *claimed* by the protocol plug-ins as well as the physical interface modules (PHY) at the relevant moments of their life in the module's buffer space. There is no explicit concept of processing hierarchy, e.g., enforcing traditional layers; thus, packets in VNETI are handled "holistically."

All API functions interfacing the application (called the *praxis* in PicOS) to VNETI have the same status as those interfacing the praxis to the kernel, i.e., they are formally system calls. As a thread in PicOS can only be resumed at a state boundary, a potentially blocking system call requires

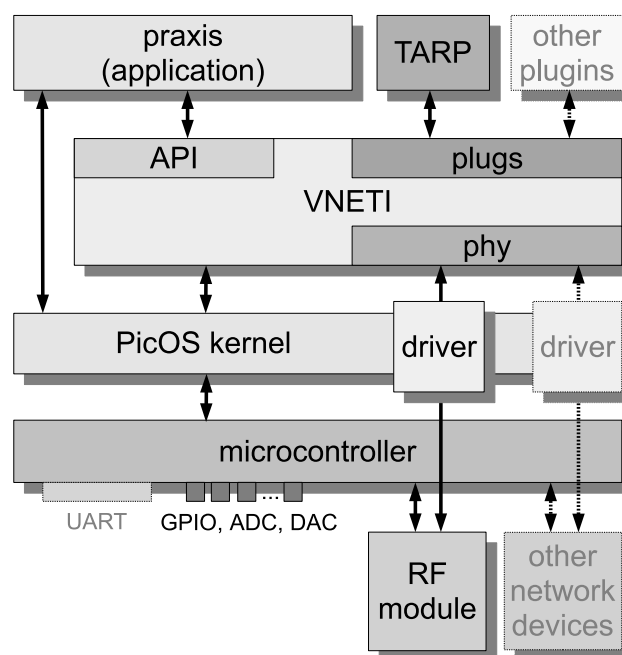
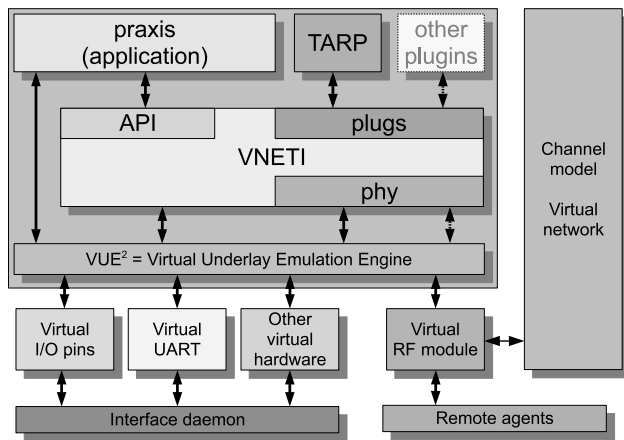


Fig. 2 The structure of PicOS

a state argument (e.g., the first argument in the function call `tcv_rnp` in Fig. 1). In this case, the function `tcv_rnp` (belonging to VNETI) is called to receive the next packet from a network session represented by descriptor `efd`. It may return immediately (if a packet is already available in the queue) or block (if the packet is yet to arrive). In the latter case, the system call will block the thread and resume it in the indicated state when it makes sense to re-execute `tcv_rnp`, i.e., upon a packet reception.

VNETI plays the role of the intermediary, without being a "layer" for reasons of efficiency as well as to open the possibility for direct interaction among networking (and other) devices. The latter is needed for implementing agile approaches to networking, such as the ones advocated by the *cross-layer design* paradigm. Note that the definition of cross-layer design, while not particularly exact, is usually associated with wireless environments because of the need to explore the fast dynamics of the wireless physical medium.

The philosophy captured by VNETI bears resemblance to various works on mechanisms to facilitate the integration and interaction of new protocol stacks. A notable example is a general protocol stack interface which has been proposed for Erlang [31]. However, few systems and language runtime libraries venture into novelty in this respect; instead, they content themselves with legacy TCP/IP layering as a matter of fact. VNETI offers a different approach that fits the needs of small sensor platforms and reflects our particular concern of avoiding overwhelming complexity in support of protocol integration mechanisms.



**Fig. 3** The structure of a VUE<sup>2</sup> model

In a traditional “layered” setting, applications are usually built to interact with the topmost layer (e.g., via a transport layer API, such as sockets or streams). Even then, interaction of an application with any intermediate layer is still provided to some degree, albeit in a second interface which is awkward and circuitous (e.g., the `ioctl()` function in Unix), thus revealing the intention of intermediate layers to interact primarily with other layers rather than directly with applications. Instead, VNETI avoids being a “layer” and acts as a module that brings together the protocols with the data items. In fact, it is fair to say that VNETI acts as a facilitator of data and event dispatching across protocol and application entities/threads.

### 3 VUE<sup>2</sup>

The close relationship between PicOS and SMURPH (Sect. 2.1) makes it possible to automatically transform PicOS praxes into SMURPH models with the intention of executing them virtually. VUE<sup>2</sup> implements the PicOS API within SMURPH, and in some cases, it directly transforms PicOS keywords into their SMURPH counterparts. To represent the physical environment of a PicOS praxis, it also provides a collection of event-driven interfaces. This way, a praxis can be compiled and executed in the environment shown in Fig. 3, with all the relevant physical elements of its node replaced by their detailed SMURPH models. Notably, the exact same source code of VNETI is used in both cases.

#### 3.1 Time flow

The fidelity of the emulation environment depends to a great extent on appropriately handling the flow of time, i.e., equating emulated time with real time. In SMURPH, as in all event-driven simulators, the time tags associated with events are purely virtual. The actual (physical) execution time of a

SMURPH thread is essentially irrelevant (unless it renders the model execution too long to wait for the results), and all that matters is the abstract delays separating the virtual events. For example, two threads in SMURPH may be semantically equivalent, even though one of them may exhibit a drastically shorter execution time than the other, e.g., due to more careful programming and/or optimization. In PicOS, however, the threads are not (just) models but they run the “real thing.” Consequently, the execution time of a thread may directly influence the perceived behavior of the PicOS node. In this context, the following two assumptions made the VUE<sup>2</sup> project worthwhile:

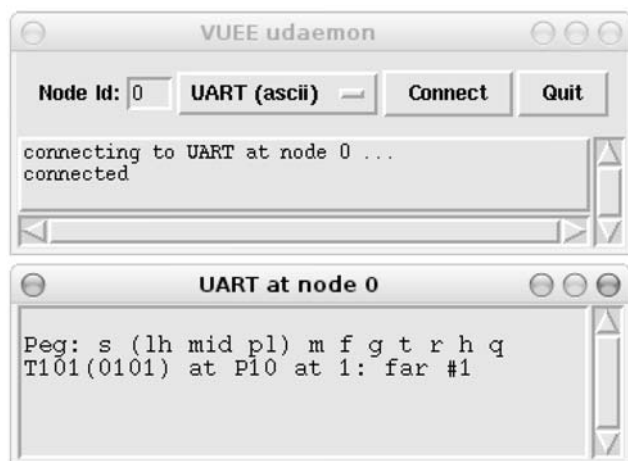
1. PicOS programs are reactive, i.e., they are practically never CPU bound. The primary reason why a PicOS thread is making no progress is that it is waiting for a peripheral event rather than the completion of some calculation.
2. If needed (from the viewpoint of model fidelity), an extensive period of CPU activity can be modeled in SMURPH by appropriately (and explicitly) delaying certain state transitions.

In most cases, we can ignore the fact that the execution of a PicOS program takes time at all and only focus on reflecting the accurate behavior of the external events. With this assumption, the job of porting a PicOS praxis to its SMURPH model can be made simple. To further increase the practical value of such a model, SMURPH provides for the so-called *visualization mode* of execution. In that mode, SMURPH tries to map the virtual time of modeled events to real time, such that the user has an impression of talking to a real application. This is only possible if the network size and complexity allow the simulator to catch up with the model execution to real time; otherwise, a suitable slow motion factor can be applied.

#### 3.2 Model scope

SMURPH threads are programmed in C++, which we have extended with new keywords and constructs. A special preprocessor (dubbed SMPP) processes the SMURPH source to produce pure C++ code. PicOS praxes are programmed in plain C with the assistance of a few macros (see Fig. 1) expanded by the standard C preprocessor. Putting trivial syntactic issues aside, the most fundamental difference between the two systems is the fact that a SMURPH model must describe the whole network (i.e., a multitude of nodes, each of them running a private copy of the application), while a complete PicOS praxis is a single program that runs on a single device. This difference becomes more pronounced if the network consists of nodes running different praxes, a not uncommon scenario.

We make extensive use of C++ classes to accomplish the conversion from a single-application node to a multi-node



**Fig. 4** A screen shot of `udaemon` showing its primary window (*top*) and interaction with an individual node over UART (*bottom*)

(and possibly multi-application) emulator. In the conversion, each PicOS praxis becomes a C++ class. Most of the praxis functions and variables become member functions and attributes of the class, respectively. For the truly global (node indifferent) functions and (constant) data, the compiler need not associate them with a specific class and can instead keep them global. When the emulator executes and builds the network, it represents each node as an object (i.e., instance of the appropriate class).

Beyond the “adaptation layer” for PicOS praxes, the VUE<sup>2</sup> extension to SMURPH implements detailed models for the physical hardware (Sect. 3.3). In terms of communication, SMURPH has a powerful generic wireless channel model [32] that provides enough flexibility to implement arbitrarily complex propagation models. All of this potential for modeling allows us to confidently and comprehensively verify applications before programming physical nodes.

### 3.3 Peripherals

The current version of VUE<sup>2</sup> implements detailed models for a significant subset of PicOS-supported peripherals that include serial communications (UART), physical sensors, general-purpose I/O (GPIO), digital-to-analog converters (DACs), analog-to-digital converters (ADCs), and light-emitting diodes (LEDs). Some of these devices, such as the GPIO pins, may require input or produce output. For such devices, VUE<sup>2</sup> offers a variety of peripheral-dependent options. In the case of the GPIO pins, the developer can describe their I/O via (a) the initial network description (for input only), (b) external files (to pre-generate/script input and log output), or (c) communication over a network socket. In the third case, VUE<sup>2</sup> provides a special program named `udaemon` that allows the developer to interactively read from and write to the peripheral.

The `udaemon` application is a fundamental component in the *interactive* emulation of a wireless sensor node and its peripherals. The initial window in the GUI (Fig. 4, top) allows the user to open peripheral-specific windows for individual emulated nodes. For example, the UART window (Fig. 4, bottom) allows two-way communication with a node over a virtual serial interface. This `udaemon` application provides access to all VUE<sup>2</sup>-modeled peripherals.

### 3.4 Operations support system

When `udaemon` connects to a node’s emulated serial port, it does so using a network socket. After a simple initialization procedure, communication proceeds indistinguishably from the serial port of a real device.

In many sensor network deployments, at least one sensor node is connected to a computer via a serial connection. This sensor node may forward all received readings to the computer for processing or archival. Software on that computer may also issue commands to the sensor network. We refer to the software running on the computer as the Operations Support System (OSS).

Slight modifications to the OSS allow it to optionally open a VUE<sup>2</sup> socket rather than a serial port to a real node, and, from that point on, communicate with the virtual node in the same way as it would with a real one. This way, the OSS can be thoroughly tested with the emulated network.

Beyond simply testing, interfacing the OSS with the emulator has proven quite useful on projects with multiple developers. A developer may be assigned components that interact with the sensor network. With the OSS connected to the emulator, the developer can work on these components without the need for real hardware.

## 4 Application development

We have used both PicOS and VUE<sup>2</sup> together to implement and test a variety of practical wireless network applications such as passively monitoring environmental conditions and actively tracking the movement of indoor objects. In the subsections that follow, we introduce a few of our applications and highlight VUE<sup>2</sup>’s ability to accommodate their specific (often peripheral-related) virtualization requirements.

### 4.1 EcoNet

The *EcoNet* project,<sup>3</sup> conceived with the Earth Observation Systems Laboratory at the University of Alberta, aims to monitor an environment’s sunlight, temperature, and humidity. Wireless sensor nodes distributed throughout an environment periodically measure these characteristics and

<sup>3</sup>See <http://econet.cs.ualberta.ca/>.

then report them to a sink node. In this scenario, a single deployment uses two separate applications: a *collector* to read/transmit sensor values and an *aggregator* to receive sensor values. The collector application uses the PicOS sensor functionality to read the current values from its analog sensors.

During execution, the collector application calls the PicOS function `read_sensor` to obtain the latest values from a device's sensors. When running on the hardware, PicOS obtains these values using the hardware's ADC. In the emulator, the user can provide the sensor values graphically on a per-node basis using slider widgets. During an experiment, simply dragging the sliders interactively changes the sensor values visible at a node.

#### 4.2 Mousetraps

The *Mousetrap* project, conceived with researchers in the University of Alberta's biology department, aims to monitor the common live-catching trap. When a rodent enters one of our traps, the movement of the ramp triggers a physical switch that we have added to the trap. The triggering of the switch creates a message that the node sends to its associated sink. In this network, nodes run a single application that uses PicOS's pin monitoring/notifier functionality for digital input.

The API for the pin notifier functionality includes functions to enable the notifier, disable it, and check its status. An application that uses the notifier will wait for the predefined event `PMON_NOTEVENT`. On the actual hardware, this event is implemented through interrupt handlers and some auxiliary functions, e.g., needed for debouncing the switch. In the emulator, we use essentially the same code, and the user can graphically change the value of a monitored input pin using a button in the GUI.

#### 4.3 Tags and Pegs

The *Tags and Pegs* project at Olsonet Communication Corporation aims to locate a sensor-enabled object within a sensor-enabled environment. Nodes in the network periodically broadcast short messages, and then other nodes use received signal strengths to perform localization. This deployment also uses two applications: one for mobile nodes and one for static nodes. To obtain signal strength values, nodes use PicOS's standard packet reception functions.

When the application calls the PicOS function `net_rx`, then it can retrieve a received packet from VNETI and, at the same time, acquire the corresponding signal strength. In the hardware, the signal strength comes from the radio transceiver. For the emulator, SMURPH's wireless model calculates signal strengths for all virtual receptions, and the virtualization of the PicOS API uses these calculated values. From the application's perspective, there is no difference between the hardware and virtual environment.

## 5 Case study: ping

In this section, we describe a simple *ping* application to illustrate how our platform transforms a single set of source files into code suitable for compilation with both the hardware (PicOS) and the emulator (VUE<sup>2</sup>). In this example, two nodes run identical copies of the software. When powered on, a node immediately begins to broadcast unaddressed ping packets that contain a locally maintained sequence number. Whenever a node receives such a ping packet, it broadcasts an acknowledgment that contains the received sequence number. Upon receiving an acknowledgement with the last sent sequence number, a node increments its locally maintained sequence number and immediately broadcasts another ping packet. If a ping packet goes unacknowledged, a node retransmits the ping after a predetermined delay.

We logically divide the ping application into three processes. In our platform, execution begins at the *thread root* (Fig. 5a, right), which is akin to the function `main` in a traditional C program. In a *strand* named `sender` (Fig. 5a, left), we place the code that transmits ping packets. We use a strand so that we can pass it the retransmission delay as an argument (in this case, line 34 sets the delay to about two seconds). Finally, we place the code for packet reception and acknowledgement generation in a *thread* named `receiver` (not shown).

Consider first the `root` thread (Fig. 5a, right). The keyword `entry` identifies a state boundary and its argument assigns a name to the state. This thread contains a single state named `RS_INIT`. The first three lines of this state essentially serve as a constructor to (a) register a physical device with VNETI, (b) register a protocol plug-in with VNETI, and (c) open a session using that device and protocol. Following some code accounting for possible error conditions, this thread continues to enable the radio's transmitter and receiver along with starting the previously introduced processes `sender` and `receiver`. Finally, the keyword `finish` terminates the `root` process.

It is quite possible for a single application to support a variety of different physical radio transceivers. Such a case might arise where a particular deployment's specific characteristics later dictate the best hardware. VNETI's abstractions make this type of flexibility possible. For each supported radio transceiver, the VNETI API provides a single function with the prefix `phys_` to register the physical device (e.g., Fig. 5a:25). To support multiple radios, developers can use trivial preprocessor directives (e.g., `#if` and `#endif`) to call the appropriate `phys_` function. Beyond this initialization stage, most applications require no further changes to switch between different transceivers.

Another noteworthy point is VNETI's protocol plug-in registration using the function `tcv_plug(...)` (e.g., Fig. 5a:26). By registering the `null` protocol plug-in

```

01: strand (sender, word)
02:   entry (SN_SEND)
03:   if (last_ack != last_snt) {
04:     delay ((word)data, SN_NEXT);
05:     when (&last_ack, SN_SEND);
06:     release;
07:   }
08:   last_snt++;
09:   proceed (SN_NEXT);
10: entry (SN_NEXT)
11:   x_packet = tcv_wnp (SN_NEXT, sfd,
12:                     DATA_LENGTH);
13:   x_packet[0] = 0;
14:   x_packet[1] = PKT_DAT;
15:   ((lword*)x_packet)[1] = wtonl (last_snt);
16:   tcv_endp (x_packet);
17: entry (SN_OUT)
18:   ser_outf (SN_OUT, "SND %lu, len = %d\r\n",
19:            last_snt, DATA_LENGTH);
20:   proceed (SN_SEND);
21: endstrand

22: thread (root)
23:   entry (RS_INIT)
24:   // setup the radio
25:   phys_dm2200 (DEV_ID, MAX_LENGTH);
26:   tcv_plug (DEV_ID, &plug_null);
27:   sfd = tcv_open (WNONE, DEV_ID, 0);
28:   if (sfd < 0) {
29:     diag ("Cannot open tcv interface");
30:     halt ();
31:   }
32:   // start sender
33:   tcv_control (sfd, PHYSOPT_TXON, NULL);
34:   runstrand (sender, 2048);
35:   // start receiver
36:   tcv_control (sfd, PHYSOPT_RXON, NULL);
37:   runthread (receiver);
38:   // done with initialization
39:   finish;
40: endthread

```

(a) User-written code for the processes **sender** and **root** prior to preprocessing.

```

P01: int sender (word zz_st, address zz_da) {
P02:   word *data = (word*) zz_da;
P03:   switch (zz_st) {
P04:   case 0:
P05:     if (last_ack != last_snt) {
P06:       delay ((word)data, 10);
P07:       zzz_uwait ((word)(&last_ack), 0);
P08:       zz_restart_entry ();
P09:     }
P10:     last_snt++;
P11:     proceed (10);
P12:   case 10:
P13:     x_packet = tcv_wnp (10, sfd, 10);
P14:     x_packet[0] = 0;
P15:     x_packet[1] = 0xABCD;
P16:     ((lword*)x_packet)[1] =
P17:       (((last_snt) & 0xffff) << 16) |
P18:       (((last_snt) >> 16) & 0xffff);
P19:     tcv_endp (x_packet);
P20:   case 20:
P21:     ser_outf (20, "SND %lu, len = %d\r\n",
P22:              last_snt, 10);
P23:     proceed (0);
P24:     break;
P25:   default:
P26:     if (zz_st == 0xffff)
P27:       return (0);
P28:     zz_badstate ();
P29:   }
P30:   return 1;
P31: }

```

(b) Preprocessed code for the process **sender** when targeting **PicOS**.

```

V01: void sender::zz_code () {
V02:   switch (TheState) {
V03:   case SN_SEND: __state_label_SN_SEND:
V04:     if (((PingNode *)TheStation)->_na_last_ack) != (((PingNode *)TheStation)->_na_last_snt)) {
V05:       ( ((PicOSNode*)TheStation)->_na_delay ((word)data, SN_NEXT) );
V06:       ( ((PicOSNode*)TheStation)->_na_when (
V07:         ((int)(IPointer) (&(((PingNode *)TheStation)->_na_last_ack))), SN_SEND) );
V08:       return;
V09:     }
V10:     (((PingNode *)TheStation)->_na_last_snt)++;
V11:     do { zz_AI_timer.zz_proceed (SN_NEXT); return; } while (0);
V12:   case SN_NEXT: __state_label_SN_NEXT:
V13:     x_packet = ( ((PicOSNode*)TheStation)->_na_tcv_wnp (
V14:       SN_NEXT, (((PingNode *)TheStation)->_na_sfd), 10) );
V15:     x_packet[0] = 0;
V16:     x_packet[1] = 0xABCD;
V17:     ((lword*)x_packet)[1] = ((((((PingNode *)TheStation)->_na_last_snt)) & 0xffff) << 16) |
V18:                             ((((((PingNode *)TheStation)->_na_last_snt)) >> 16) & 0xffff));
V19:     ( ((PicOSNode*)TheStation)->_na_tcv_endp (x_packet) );
V20:   case SN_OUT: __state_label_SN_OUT:
V21:     ( ((PicOSNode*)TheStation)->_na_ser_outf (SN_OUT, "SND %lu, len = %d\r\n",
V22:       (((PingNode *)TheStation)->_na_last_snt), 10) );
V23:     do { zz_AI_timer.zz_proceed (SN_SEND); return; } while (0);
V24:   }
V25: }

```

(c) Preprocessed code for the process **sender** when targeting **VUE<sup>2</sup>**.**Fig. 5** Excerpts from the ping application's source code both before and after preprocessing

**Table 1** Some of the most common VNETI API functions and their descriptions

Function	Description
<code>tcv_plug</code>	Configures a protocol plug-in for the network interface; in the ping application, the null plug-in provides a more or less direct connection to the network
<code>tcv_open</code>	Opens a session and returns a session descriptor (akin to a file descriptor)
<code>tcv_control</code>	Allows the application to change various parameters associated with the transceiver; in the ping application, we use it to enable the transmit and receive functionality of the radio
<code>tcv_rnp</code>	Acquires the next packet queued for reception at the session
<code>tcv_wnp</code>	Requests a packet handle from VNETI in order to send a new outgoing packet
<code>tcv_left</code>	Returns the length of a packet acquired by <code>tcv_rnp</code> or the bytes remaining to fill in an outgoing packet's buffer
<code>tcv_endp</code>	Indicates explicitly the moment when a packet has been processed and is no longer needed

for the ping application, calls to functions in the VNETI API provide the programmer with a more or less direct connection to the network. The programmer then has much flexibility to manage the packet overhead. Beyond the *null* plug-in, our platform also implements the Tiny Ad hoc Routing Protocol (TARP) to perform the ad hoc routing mentioned in the introduction. Given our plug-in oriented approach, users can implement further protocols as desired.

The remaining VNETI API functions begin with the prefix `tcv_` and primarily serve as state and buffer management. Table 1 briefly describes some of the `tcv_` functions relevant to the presented code.

In Fig. 5a, left, we present the code for the strand `sender`. In our platform, we define a number of data types to provide consistent variable sizes between the different targets. The keyword `word` that appears in the definition of `sender` identifies the type of its *data* argument (the `word` type is a 16-bit unsigned value). Later in the strand, the user can access this data argument using the (implicit) variable `data`. Upon entering the state `SN_SEND`, the program checks whether the node has received an acknowledgement for the last ping. If so, it increments the sequence number and immediately proceeds to send another ping. If not, it (a) sets a timer to delay before rebroadcasting the ping and (b) waits for a signal (IPC) on the address of `last_ack`. The receiver process (not shown) triggers the address of `last_ack` when it receives an expected acknowledgement. By waiting for this signal in `sender`, the application

can immediately react to a received acknowledgement by advancing the sequence number and sending out a new ping packet.

The programmer's effort amounts to writing code similar to that presented in Fig. 5a. Note that the context for this code is a single node and the programming language is plain C, albeit enhanced with new "keywords" (to improve clarity and simplify programming) that we have implemented as preprocessor macros. Since the code is plain C, compiling for PicOS simply uses the standard C preprocessor and compiler. For VUE<sup>2</sup>, a specialized preprocessor makes the more complicated transition to C++ code where multiple applications and nodes must operate in a single simulation environment which preserves the independent nature of those individual nodes.

## 5.1 Preprocessing

In Fig. 5b and c, we show the changes made to `sender` by the respective preprocessor to prepare the code for compilation with PicOS and VUE<sup>2</sup>. In this subsection, we describe some of the changes along with the reasoning behind them.

The first thing to notice is that in both cases the code's general structure remains the same. The user-written finite state machine using our `strand/entry` "keywords" becomes a switch on a variable containing the current state. In the PicOS case, the preprocessor substitutes state names with integer constants (because the labels are `#define` preprocessor directives). For VUE<sup>2</sup>, the symbolic names remain because an enumerated type represents states and thus the compiler introduces the integer constants rather than the preprocessor.

In the VUE<sup>2</sup> code, notice the appearance of several new variables (i.e., `TheStation` and `TheState`). These variables (and others) arise in the transition from a single-node (hardware) environment to a multi-node (simulated) environment. The simulator represents each node in the network as an object, and the variable `TheStation` points to the node currently being simulated. Another global variable named `TheProcess` identifies the current thread (e.g., `sender`) within the current node. Finally, a global variable named `TheState` holds an integer that identifies the current state within that process. At any point, these three variables collectively describe the current state of the simulation.

Notice that all accesses to system calls and node-specific variables within the user's application use the pointer `TheStation` (e.g., Fig. 5c:V04-V07,V10). At different places in the preprocessed code, the single object is typecast to either a `PingNode` or a `PicOSNode`. The derivation of the relevant classes is as follows. SMURPH provides a base class to represent a virtual hardware frame (a chassis of sorts) describing a processing unit of the network model.

This class is named `Station`. Building on this concept, VUE<sup>2</sup> then introduces the notion of a `PicOSNode` as a specialized type of station and thus derives it from `Station`. At this level, VUE<sup>2</sup> defines the PicOS system calls (including those for VNETI) and internal state variables. From here, VUE<sup>2</sup> derives a further class that is protocol plug-in specific; it contains the functions and variables necessary to implement the plug-in. In this case, we use the `null` protocol plug-in and thus this further subclass is of the type `NNode`. Note that VUE<sup>2</sup> also provides a class `TNode` for nodes that run the *TARP* protocol plug-in. Finally, the class representing the actual application (in this case `PingNode`) inherits from the protocol-specific class (in this case `NNode`) and further defines the processes and variables of the user's application. When starting a simulation run, VUE<sup>2</sup> builds all of the network nodes using the lowest-level (and most complete) class, which in this case is `PingNode`. To see the different typecasts, first look to line V04, which typecasts to `PingNode` when accessing node-private application variables, and then to lines V05 and V06, which typecast to `PicOSNode` for making system calls.

Notice that the C++ version contains additional labels on lines V03, V12, and V20 that begin with the text `__state_label_`. When the simulator comes across the keyword `proceed` (e.g., expanded in lines V11 and V23), it saves the next state in the variable `TheState` and then returns control to the scheduler to make the state transition. The scheduler may not immediately return control to the process if other events occur at the same time. When the process does resume, the `switch` statement on the variable `TheState` moves the process into the appropriate state. Sometimes, a SMURPH thread may want to make a transition that does not involve the scheduler. In such a case, using the command `sameas` (not shown) rather than `proceed` accommodates an immediate transition using the label `__state_label_` along with a `goto` statement. Note that none of the code written for VUE<sup>2</sup>/PicOS takes advantage of that functionality (it is neither required nor natural in PicOS).

Readers may be unfamiliar with the construct

```
do ... while (0);
```

introduced on lines V11 and V23. It results from a macro expansion of our keyword `proceed`. Essentially, this code is a C idiom to define a macro consisting of multiple statements that has the syntactic rights of a single statement.

The preprocessing that takes place both in PicOS as well as in VUE<sup>2</sup> involves some name mangling. In PicOS, the characters `zzz_`, `zz_`, or `x_` are prepended to the names of intentionally internal variables and function in an attempt to avoid accidental conflict with user-introduced names. In VUE<sup>2</sup>, additional mangling is applied to cover remote access to node attributes into local-looking references to simple variables.

## 6 Conclusion

In this paper, we described our “holistic” platform for building wireless ad hoc sensor networks and focused on its most representative and essential component: VUE<sup>2</sup> (the Virtual Underlay Emulation Engine). Using it, developers can write applications in C, rather than an invented programming language or bytecode, and then easily target both hardware nodes and our emulator.

Through the development of several applications, we have found that the finite state machine paradigm allows for the natural representation reactive applications. By using VUE<sup>2</sup> during the development stage, we can test our applications, both sensor node and operations support system software, exhaustively in a virtual environment before investing time to program physical hardware. When we later move our applications to the hardware, they perform within our expectations.

**Acknowledgements** The authors would like to thank the Natural Sciences and Engineering Research Council (NSERC) and the Informatics Circle of Research Excellence (iCORE) for helping fund this work.

## References

- Perkins, C., & Bhagwat, P. (1993). Highly dynamic Destination-Sequenced Distance Vector routing (DSDV) for mobile computers. In *Proc. of SIGCOMM'94* (pp. 234–244), Aug. 1993.
- Chen, T.-W., & Gerla, M. (1998). Global state routing: a new routing scheme for ad-hoc wireless networks. In *Proc. of ICC'98*, June 1998.
- Perkins, C., Royer, E. B., & Das, S. (2003). Ad-hoc On-demand Distance Vector routing (AODV). February 2003, Internet Draft: [draft-ietf-manet-aodv-13.txt](#).
- Li, J., Jannotti, J., Couto, D. D., Karger, D., & Morris, R. (2000). A scalable location service for geographic ad hoc routing. In *Proc. of the ACM/IEEE intl. conference on mobile computing and networking (MOBICOM'00)* (pp. 120–130).
- Johnson, D. B., & Maltz, D. A. (1996). Dynamic Source Routing in ad hoc wireless networks. In K. Imielinski, & K. Korth (Eds.), *Mobile computing* (Vol. 353). Amsterdam: Kluwer Academic.
- Park, V., & Corson, M. (1998). A performance comparison of TORA and ideal link state routing. In *Proc. of IEEE symposium on comp. and comm.*, June 1998.
- Toh, C.-K. (1996). A novel distributed routing protocol to support ad-hoc mobile computing. In *Proc. of IEEE 15th annual intl. phoenix conf. on comp. and comm.* (pp. 480–486), Mar. 1996.
- Gunes, M., Sorges, U., & Bouazizi, I. (2002). ARA—the ant-colony based routing algorithm for MANETS. In *Proceedings of international workshop on Ad-hoc networking (IWAHN)*, Vancouver, British Columbia, Canada, Aug. 2002.
- Henderson, T., Park, J., Smith, N., & Wright, R. (2003). From motes to Java stamps: Smart sensor network testbeds. In *Intelligent robots and systems* (pp. 799–804), Las Vegas, NV, Oct. 2003.
- Levis, P., & Culler, D. (2002). Maté: A tiny virtual machine for sensor networks. In *Proc. of the 10th intl. conference on architectural support for programming languages and operating systems (ASPLOS-X)* (pp. 85–95), San Jose, CA, Oct. 2002.

11. Akhmetshina, E., Gburzyński, P., & Vizeacoumar, F. (2003). PicOS: A tiny operating system for extremely small embedded platforms. In *Proc. of ESA'03* (pp. 116–122), Las Vegas, Jun. 2003.
12. Hui, J. (2004). *TinyOS network programming* (version 1.0). TinyOS 1.1.8 Documentation.
13. Levis, P. et al. (2005). TinyOS: An operating system for sensor networks. In W. Weber, J. Rabaey, & E. Aarts (Eds.), *Ambient intelligence* (pp. 115–148). Berlin: Springer.
14. Dahl, O., & Nygaard, K. (1967). *Simula: A language for programming and description of discrete event systems*. Norwegian Computing Center, Oslo, Introduction and user's manual, 5th edition.
15. Birthwistle, G., Dahl, O., Myrhaug, B., & Nygaard, K. (1973). *Simula begin*. Oslo: Studentlitteratur.
16. Gustafsson, A. (2005). Threads without the pain. *Social Computing*, 3(9), 34–41.
17. Thomas, D., Fowler, C., & Hunt, A. (2004). *Programming ruby: The pragmatic programmer's guide* (2nd edn.). The Pragmatic Programmers.
18. Gburzyński, P. (1996). *Protocol design for local and metropolitan area networks*. New York: Prentice-Hall.
19. Dobosiewicz, W., & Gburzyński, P. (1997). Protocol design in SMURPH. In J. Walrand, & K. Bagchi (Eds.), *State-of-the-art in performance modeling and simulation* (pp. 255–274). New York: Gordon and Breach.
20. Dobosiewicz, W., & Gburzyński, P. (1991). On the apparent unfairness of a capacity-1 protocol for very fast local area networks. In *Proceedings of the third IEE conference on telecommunications*, Edinburgh, Scotland, Mar. 1991.
21. Dobosiewicz, W., & Gburzyński, P. (1994). An alternative to FDDI: DPMA and the pretzel ring. *IEEE Transactions on Communications*, 42, 1076–1083.
22. Dobosiewicz, W., & Gburzyński, P. (1997). The spiral ring. *Computer Communications*, 20(6), 449–461.
23. Molle, M. (1994). *A new binary logarithmic arbitration method for Ethernet*. Computer Systems Research Institute, Toronto, Ontario, Canada, CSRI-298.
24. Berard, M., Gburzyński, P., & Rudnicki, P. (1991). Developing MAC protocols with global observers. In *Proceedings of computer networks '91* (pp. 261–270), Jun. 1991.
25. Altisen, K., Maranchini, F., & Stauch, D. (2005). *Aspect-oriented programming for reactive systems: a proposal in the synchronous framework* (Verimag CNRS, Research Report #TR-2005-18), Nov. 2005.
26. Yartsev, B., Korneev, G., Shalyto, A., & Ktov, V. (2005). Automata-based programming of the reactive multi-agent control systems. In *Intl. conference on integration of knowledge intensive multi-agent systems* (pp. 449–453), Waltham, MA, Apr. 2005.
27. Gburzyński, P., & Maitan, J. (1997). Simulation and control of reactive systems. In *Proceedings of winter simulation conference WSC'97* (pp. 413–420), Atlanta, Georgia, Dec. 1997.
28. Gburzyński, P., Maitan, J., & Hillyer, L. (1998). Virtual prototyping of reactive systems in SIDE. In *Proceedings of the 5th European concurrent engineering conference ECEC'98* (pp. 75–79), Erlangen-Nuremberg, Germany, Apr. 1998.
29. Olesinski, W., Rahman, A., & Gburzyński, P. (2003). TARP: a tiny ad-hoc routing protocol for wireless networks. In *Australian telecommunication, networks and applications conference (ATNAC)*, Melbourne, Australia, Dec. 2003.
30. Gburzyński, P., Kaminska, B., & Olesinski, W. (2007). A tiny and efficient wireless ad-hoc protocol for low-cost sensor networks. In *Proc. of design automation and test in Europe (DATE'07)* (pp. 1562–1567), Nice, France, Apr. 2007.
31. Anderson, P., & Kvisth, M. (2000). *A general protocol stack interface in Erlang*. Master's thesis, Uppsala University, Department of Computing Science.
32. Gburzyński, P., & Nikolaidis, I. (2006). Wireless network simulation extensions in SMURPH/SIDE. In *Proc. of the 2006 winter simulation conference (WSC'06)*, Monterey, California, Dec. 2006.



ing them in every day activities, and enhancing their independence. Nicholas's work deals with software development for tiny devices, software specification, real-time systems, simulation, and performance evaluation.



ations) and TARP (a resilient ad-hoc routing scheme for low-cost sensor networks). He has devised a number of communication protocols, including ad-hoc wireless routing schemes and custom protocols for high-performance systems, and implemented many practical wireless networking solutions. His research interests are in telecommunication, embedded systems, operating systems, simulation and performance evaluation. As a hobby, he develops anti-spamming tools (<http://sfm.cs.ualberta.ca>).



**Nicholas M. Boers** received his B.Sc. from Malaspina University-College in 2004 and his M.Sc. in Computing Science from the University of Alberta in 2006. He is currently working towards a Ph.D. at the University of Alberta. He has been involved in several projects related to wireless networking, including ad-hoc sensor networks, most notably the Smart Condo project aimed at non-intrusively monitoring elderly and disabled people for the purpose of diagnosing medical problems, assisting them in every day activities, and enhancing their independence. Nicholas's work deals with software development for tiny devices, software specification, real-time systems, simulation, and performance evaluation.

**Paweł Gburzyński** holds a Ph.D. in Computer Science from the University of Warsaw, Poland. Since 1985 he has been with the faculty of the Department of Computing Science, University of Alberta, where he is a professor. In 2002, he co-founded Olsonet Communications (<http://www.olsonet.com>), where he acts as chief scientist. Dr. Gburzyński's contributions include SMURPH (a high-fidelity modeling/emulation package for communication systems), PicOS (an operating system for embedded applications) and TARP (a resilient ad-hoc routing scheme for low-cost sensor networks). He has devised a number of communication protocols, including ad-hoc wireless routing schemes and custom protocols for high-performance systems, and implemented many practical wireless networking solutions. His research interests are in telecommunication, embedded systems, operating systems, simulation and performance evaluation. As a hobby, he develops anti-spamming tools (<http://sfm.cs.ualberta.ca>).

**Ioanis Nikolaidis** is a Professor with the Computing Science Department at the University of Alberta. He received his B.Sc. from the University of Patras, Greece, in 1989 and his M.Sc. and Ph.D. in Computer Science from Georgia Tech in 1991 and 1994, respectively. Between 1994 and 1996 he worked for the European Computer Industry Research Center in Munich, Germany, in the area of distributed computing. He joined the University of Alberta in January 1997. He has published more than

seventy articles in books, journals, and conference proceedings in the area of computer networking. His research interest range from network modeling and simulation, to large scale data delivery systems, to mobile and secure networking. He served for ten years (1999–2009) in the editorial board of the Computer Networks journal (Elsevier). Since 1999 he has been a member of the editorial board (and served as Editor in Chief between 2007 and 2009) of the IEEE Network magazine. He has served in the technical program committees of numerous conferences, including ICC, Globecom, INFOCOM, LCN, IPCCC, PerCom, IFIP Networking, and CNSR. He is in the steering committee of WLN (co-located annually with IEEE LCN) and in the steering committee of the ADHOCNOW conference. He is currently the Graduate Admissions Chair of the Computing Science Department at the University of Alberta.



**Wlodek Olesiński** (Partner, President & CEO) co-founded Olsonet Communications Corporation, Canada, in 2002. Prior to starting the company, he worked within R&D organizations at Alcatel, Newbridge, Nortel Networks, and Bell-Northern Research, taking part in software development for PSTN and packet networks. His areas of expertise extend over network management, protocol design, operating systems, embedded systems, real-time systems, and mission-critical networks. He is holding an M.Sc. in informatics from the Jagiellonian University of Kraków, Poland. He has lived and worked in Canada, Germany, and Poland.