Proceedings of the International Multiconference on Computer Science and Information Technology pp. 853–860
ISBN 978-83-60810-14-9
ISSN 1896-7094

Supporting Wireless Application Development via Virtual Execution

Nicholas M. Boers, Pawel Gburzyński, Ioanis Nikolaidis
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{boers,pawel,yannis}@cs.ualberta.ca

Wlodek Olesinski
Olsonet Communications
51 Wycliffe Street
Ottawa, Ontario, Canada K2G 5L9
wlodek@olsonet.com

***Abstract*—We introduce our "holistic" platform for building wireless ad hoc sensor networks and focus on its most representative and essential virtualization component: VUE^2 (the Virtual Underlay Emulation Engine). Its role is to provide a vehicle for authoritative emulation of complete networked applications before physically deploying the 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^2 achieves this goal owing to several facilitating factors, most notably the powerful programming paradigm adopted in our platform. As implied by the holistic nature of the discussed system, our presentation touches upon operating systems, simulation, network protocols, real-time systems, and programming methodology.**

I. Introduction

ALTHOUGH simple wireless devices built from low-end components are quite popular these days, one seldom hears about serious wireless networks within this framework. While it is not a big challenge to implement simple broadcasters of short packets, it is quite another issue to turn them into collaborating nodes of a serious ad hoc wireless system. Apparently, many popular ad hoc routing schemes proposed and analyzed in the literature [1]–[6] address devices with a somewhat larger resource base. To make matters worse, some people believe that such devices must be programmed in Java to make serious applications possible [7].

In this context, efforts to introduce an order and methodology into programming small devices often meet with skepticism and shrugs, the common misconception being that one will not have to wait for long before those devices disappear and become superseded by larger ones capable of supporting "serious" programming platforms. This is not true. 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. On the contrary: their low cost and power requirements enable new applications and narrow the gap between the publicized promise of ad hoc wireless sensor networking and the actual state of affairs. 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.

The goal of our platform is to enable the rapid creation of 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. To thoroughly test and evaluate our work, we turn to virtualization.

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 [8]. 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 [9], part of the Google Android platform [10]) 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 at the level of the API. Unfortunately, commodity OS APIs are very broad, and the abstractions that 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 even implement 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 devices [11]) and VUE^2 (the Virtual Underlay Emulation Engine for realistically simulating networked applications programmed in PicOS).

II. PICOS

The most serious problem with implementing non-trivial, structured, multitasking software on microcontrollers with limited RAM is minimizing the amount of memory resources 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.

PicOS strikes a compromise between the complete lack of threads and overtaxing the tiny amount of RAM with 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 amount 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 threads, which are structured like finite state machines (FSMs) and exhibit the dynamics of coroutines [12], [13] 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., "stackless" Python [14] and more recently Ruby [15]). 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, because a CPU-bound task can be broken down into a sequence of states to allow preemption at state transitions. However, an important element of our view is to use the coroutine paradigm as a means to *discourage* CPU-intensive tasks on sensors. 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.

On a historical note, we arrived at PicOS indirectly as a step in the evolution of our network simulation system SMURPH [16], [17]. As SMURPH underwent a number of enhancements, 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 [18], [19].

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 self-documenting program strictly conforming to its SMURPH model.

A. The anatomy of a PicOS thread

Fig. 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.

```
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 thread can lose the CPU 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 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 IPC and `delay` for timed events). The collection of wait requests issued by a thread in every state describes the dynamic options for its transition function from that state.

B. 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 them 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 [20], [21]) 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."

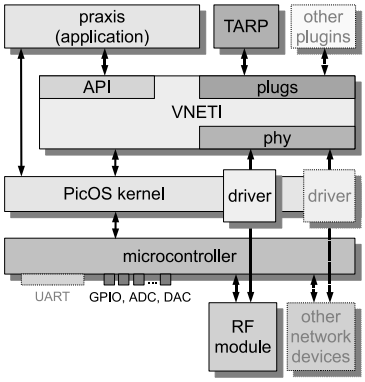

Fig. 2: The structure of PicOS.

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 a state argument (e.g., the first argument in the function call `tcv_rnp` in Fig. 1).

III. VUE^2

The close relationship between PicOS and our discrete-time event-driven network simulator named SMURPH [16], [17] makes it possible to automatically transform PicOS praxes into SMURPH models with the intention of executing them virtually. VUE^2 implements the PicOS API within SMURPH, and in some cases, it can simply transform 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, exactly the same source code of VNETI is used in both cases.

A. 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

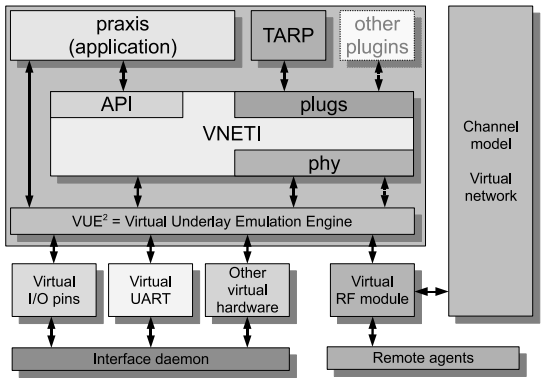

Fig. 3: The structure of a VUE^2 model.

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^2 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 VUE^2 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.

B. 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 syn-

tactic 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 (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 truly global (node indifferent) functions and 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^2 extension to SMURPH implements detailed models for the physical hardware (Section III-C). In terms of communication, SMURPH brings in a powerful generic wireless channel model [22] 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 uploading the code to physical nodes.

C. Peripherals

The current version of VUE^2 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^2 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^2 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^2-modeled peripherals.

IV. Application Development

We have used both PicOS and VUE^2 together to implement and test a variety of practical wireless network applications

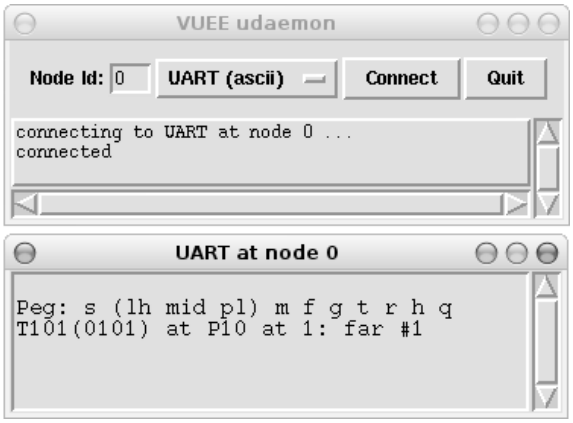

Fig. 4: A screen shot of `udaemon` showing its primary window (top) and interaction with an individual node over UART (bottom).

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^2's ability to accommodate their specific (often peripheral-related) virtualization requirements.

A. EcoNet

The *EcoNet* project, 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 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.

B. 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 the above 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, disable, and check it. An application that uses it 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.

C. Tags and Pegs

The *Tags and Pegs* project at Olsonet 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`, the application can retrieve a received packet from VNETI, and at the same time, the corresponding signal strength. In the hardware, the signal strength comes from the radio transceiver. For the emulator, SMURPH's generic 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.

V. 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^2). 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.

Before presenting code, the term *thread* (used loosely in Section II) requires some clarification. In the context of the source code, our platform makes a distinction between processes with and without arguments. By doing so, we can improve compatibility between the hardware and emulator targets. We call a process that expects a typed data argument on initialization a `strand`. Many instances of a `strand` may exist at any given time, where each operates on its own local (private) data. We call a process that tends to operate on global data and does not accept such an argument a `thread`. Only one instance of a `thread` may exist at any given time.

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 thread named `root` (Fig. 5a, right). The keyword `entry` identifies a state boundary and its argument identifies the state name. 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. After some error handling code, this thread continues to enable the radio's transmitter and receiver along with starting the previously introduced processes `sender` and `receiver`.

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 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 that we mentioned in our 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 I 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`, code checks whether the node received an acknowledgement for the last ping. If so, it immediately proceeds to send another ping. If not, it (a) sets a timer to delay before rebroadcasting the ping and (b) waits for

*The application's complete source code is available online at http://tinyurl.com/67a4t6.

```
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^2**.

Fig. 5: Excerpts from the ping application's source code both before and after preprocessing.

TABLE I: Some of the most common VNETI API functions and their descriptions.

Function	Description
`tcv_plug`	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.
`tcv_open`	Opens a session and returns a session descriptor (akin to a file descriptor).
`tcv_control`	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.
`tcv_rnp`	Acquires the next packet queued for reception at the session.
`tcv_wnp`	Requests a packet handle from VNETI in order to send a new outgoing packet.
`tcv_left`	Determines the length of a packet acquired by `tcv_rnp`.
`tcv_endp`	Indicates explicitly the moment when a packet has been processed and is no longer needed.

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 then immediately advance the sequence number and send 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 it is plain C code, 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^2, a specialized preprocessor makes the more complicated transition to C++ code where multiple applications and nodes must operate in a single simulation environment that preserves the state of those individual nodes.

A. 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^2. 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^2, 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^2 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 process (e.g., `sender`) within that node that the simulator is evaluating. 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 piece of hardware running in the network named `Station`. VUE^2 then introduces the notion of a `PicOSNode` as a specialized type of station and thus derives it from `Station`. At this level, VUE^2 defines the PicOS system calls (including those for VNETI) and internal state variables. From here, VUE^2 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^2 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, VUE^2 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., 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, SMURPH users will want to make a transition that does not involve the scheduler. In these cases, 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^2/PicOS can currently make use of the `sameas` functionality.

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.

In both the PicOS and VUE^2 preprocessed code, name mangling occurs. In PicOS, preprocessing appends the characters `zzz_`, `zz_`, or `x_` to functions and variables internal to PicOS in an attempt to avoid conflicts. In VUE^2, some similar mangling occurs plus further mangling on node-private variables where the processor appends `_na_` for similar reasons.

VI. 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^2 (the Virtual Underlay Emulation Engine). Using it, developers can write applications in C, rather than a new programming language or bytecode, and then easily target to 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^2 during the development stage, we can test our applications 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.

Acknowledgment

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 research.

References

[1] C. Perkins and P. Bhagwat, "Highly dynamic Destination-Sequenced Distance Vector routing (DSDV) for mobile computers," in *Proc. of SIGCOMM'94*, Aug. 1993, pp. 234–244.

[2] T.-W. Chen and M. Gerla, "Global state routing: a new routing scheme for ad-hoc wireless networks," in *Proc. of ICC'98*, June 1998.

[3] V. Park and M. Corson, "A performance comparison of TORA and ideal link state routing," in *Proc. of IEEE Symposium on Comp. and Comm.*, June 1998.

[4] J. Li, J. Jannotti, D. D. Couto, D. Karger, and R. Morris, "A scalable location service for geographic ad hoc routing," in *Proc. of the ACM/IEEE Intl. Conference on Mobile Computing and Networking (MOBICOM' 00)*, 2000, pp. 120–130.

[5] C. Perkins, E. B. Royer, and S. Das, "Ad-hoc On-demand Distance Vector routing (AODV)," February 2003, Internet Draft: draft-ietf-manet-aodv-13.txt.

[6] D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in ad hoc wireless networks," in *Mobile Computing*, Imielinski and Korth, Eds. Kluwer Academic Publishers, 1996, vol. 353.

[7] T. Henderson, J. Park, N. Smith, and R. Wright, "From motes to Java stamps: Smart sensor network testbeds," in *Intelligent Robots and Systems*, Las Vegas, NV, Oct. 2003, pp. 799–804.

[8] P. Levis and D. Culler, "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)*, San Jose, CA, Oct. 2002, pp. 85–95.

[9] [Online]. Available: http://www.dalvikvm.com/

[10] [Online]. Available: http://code.google.com/android/

[11] E. Akhmetshina, P. Gburzyński, and F. Vizeacoumar, "PicOS: A tiny operating system for extremely small embedded platforms," in *Proc. of ESA'03*, Las Vegas, Jun. 2003, pp. 116–122.

[12] O. Dahl and K. Nygaard, "Simula: A language for programming and description of discrete event systems," Norwegian Computing Center, Oslo, Introduction and user's manual, 5th edition, 1967.

[13] G. Birthwistle, O. Dahl, B. Myhrhaug, and K. Nygaard, *Simula Begin*. Oslo: Studentlitteratur, 1973.

[14] A. Gustafsson, "Threads without the pain," *Social Computing*, vol. 3, no. 9, pp. 34–41, 2005.

[15] D. Thomas, C. Fowler, and A. Hunt, *Programming Ruby: The Pragmatic Programmer's Guide*. The Pragmatic Programmers, 2004, second edition.

[16] P. Gburzyński, *Protocol Design for Local and Metropolitan Area Networks*. Prentice-Hall, 1996.

[17] W. Dobosiewicz and P. Gburzyński, "Protocol design in SMURPH," in *State-of-the-art in Performance Modeling and Simulation*, J. Walrand and K. Bagchi, Eds. Gordon and Breach, 1997, pp. 255–274.

[18] K. Altisen, F. Maraninchi, and D. Stauch, "Aspect-oriented programming for reactive systems: a proposal in the synchronous framework," Verimag CNRS, Research Report #TR-2005-18, Nov. 2005.

[19] B. Yartsev, G. Korneev, A. Shalyto, and V. Ktov, "Automata-based programming of the reactive multi-agent control systems," in *Intl. Conference on Integration of Knowledge Intensive Multi-Agent Systems*, Waltham, MA, Apr. 2005, pp. 449–453.

[20] W. Olesinski, A. Rahman, and P. Gburzyński, "TARP: a tiny ad-hoc routing protocol for wireless networks," in *Australian Telecommunication, Networks and Applications Conference (ATNAC)*, Melbourne, Australia, Dec. 2003.

[21] P. Gburzyński, B. Kaminska, and W. Olesinski, "A tiny and efficient wireless ad-hoc protocol for low-cost sensor networks," in *Proc. of Design Automation and Test in Europe (DATE'07)*, Nice, France, Apr. 2007, pp. 1562–1567.

[22] P. Gburzyński and I. Nikolaidis, "Wireless network simulation extensions in SMURPH/SIDE," in *Proc. of the 2006 Winter Simulation Conference (WSC'06)*, Monetery, California, Dec. 2006.