PICOS & VNETI
Enabling Real Life Layer-Less WSN Applications

Nicholas M. Boers[1], Ioanis Nikolaidis[2], Pawel Gburzynski[3], Wlodek Olesinski[3]
[1]*Department of Computer Science, Grant MacEwan University, 10700 104 Ave. NW, Edmonton, Alberta T5J 4S2, Canada*
[2]*Department of Computing Science, University of Alberta, 2-21 Athabasca Hall, Edmonton, Alberta T6G 2E8, Canada*
[3]*Olsonet Communications Corporation, 51 Wycliffe Street, Ottawa, Ontario K2G 5L9, Canada*
boersn@macewan.ca, nikolaidis@ualberta.ca, {pawel, wlodek}@olsonet.com

Keywords: Wireless Sensor Networks: Operating Systems: Programming Interfaces

Abstract: In the simple devices used for wireless sensor networks, the costs associated with a layered approach can be significant. Small-footprint operating systems have been developed by adopting non-traditional approaches to network abstractions while still aiming to simplify software development. In these approaches, some elements of modularity are valuable to retain, e.g., packet buffer management, which can be factored out of the layers and supported by a generic interface. In this paper, we describe the PicOS operating system with its versatile network interface (VNETI) and describe our experience using it. VNETI's approach to the problem, where it acts as a mediator between (a) the application programming interface, (b) protocol plug-ins, and (c) a physical input/output module, allows for an effective component-based design with low overheads. With our essentially layer-less approach to networking, we have found it intuitive to incorporate even the simplest devices into non-trivial networks.

1 INTRODUCTION

Networks are traditionally built around a stack of layers, where each layer provides a set of services via a clearly defined interface. By isolating each layer and assigning it clear responsibilities, traditional approaches achieve an overall reduction in design complexity at the cost of increased resource usage. In general-purpose computing environments with diverse and powerful machines and operating systems, the costs of layering are negligible. The situation is different for platforms with scarce memory and processing resources, where the costs of layering tend to outweigh its benefits. If one is to avoid layers in these devices, it is still important to devise an OS structure that still allows for flexible application development.

Researchers have developed a wide range of operating systems capable of running on low-power devices, e.g., (Dunkels et al., 2004; Levis et al., 2005; Abrach et al., 2003; Akhmetshina et al., 2003; Beutel, 2006; Eswaran et al., 2005), with the most popular of them being TinyOS (Levis et al., 2005). We focus on a mature and evolving alternative named PicOS (Akhmetshina et al., 2003) that has a number of advantages over the former, most notably (a) all of the program dynamics available to the programmer are captured by PicOS's threads (finite state machines) rather than interrupt service routines (or callbacks), allowing all threads to share the same (global) stack and (b) support for flexible dynamic memory allocation, even within less than 1 KB of RAM.

In this paper, we address how to structure a layer-less WSN operating system such that it still allows for modular application development. Section 2 describes the architecture of the operating system. It explores the versatile network interface, VNETI, a single meta-driver that mediates between the application programming interface (API), protocol plug-ins, and physical input/output (I/O) modules. In Section 3, we describe an application built in this framework, our experience running it in an unfriendly environment, and how that experience led to an improvement of our layer-less communication scheme. Section 4 summarizes our work with conclusions.

2 PICOS

The primary problem with implementing classical multitasking within limited RAM is minimizing the amount of per-process fixed memory resources, most notably, stack space. PicOS solves this prob-

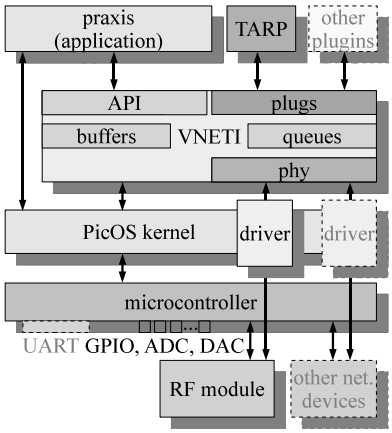

Figure 1: The relationship between VNETI and other system components.

lem by implementing a non-classical flavor of multitasking inherited from a specification/simulation environment named SIDE (Gburzynski, 1995; Gburzynski and Nikolaidis, 2006). A program's multiple tasks share the same global stack and act as co-routines with multiple entry points and implicit control transfer. A task looks like a finite state machine (FSM) that navigates its states in response to events, and the CPU is multiplexed among the multiple tasks, but only at state boundaries. This simplifies – to the point of practically eliminating – all synchronization problems within the application, while still providing a reasonable degree of concurrency and responsiveness.

Since WSN applications are predominantly reactive, i.e., not CPU bound, it is quite natural to express them as FSMs. Although many types of applications can be expressed as FSMs, the format is especially useful and natural for reactive applications that respond to possibly complicated configurations of events. While one FSM remains blocked, other FSMs can continue to operate independently.

The operating system provides a wide variety of library functions as well as tools for interacting with peripherals. A key software component (VNETI) mediates and standardizes interaction with the radio transceiver and (optionally) the universal asynchronous receiver/transmitter (UART) (Figure 1).

2.1 VNETI

The purpose of the Versatile NETwork Interface (VNETI) is to provide a simple collection of APIs, independent of the underlying I/O driver implementation, which, in addition to enabling the rapid deployment of networked applications, make it easy to develop testbeds using emulated I/O interfaces. To avoid potentially problematic protocol layering in small footprint solutions, the presented interface is essentially layer-less and its semi-complete generic functionality can be redefined by plug-ins.

The actual implementation of the physical interface can be encapsulated as a relatively simple and easily exchangeable module. In a drastic departure from the layered approach, the plug-ins facilitate modularity and incorporate functionality that would, conceptually, span across many layers in a traditional layered design. For example, there is no restriction preventing plug-ins from consulting the "payload" as well as any "descriptive" information present in a packet, i.e., headers. Multiple plug-ins and physical interfaces can coexist within the same system configuration.

VNETI (Figure 1) implements (a) transparent management of buffer (packet) storage in a dynamic number of queues with per-packet timeouts, (b) multiple application access points, and (c) a unified set of functions for interfacing plug-ins and physical modules. It acts as a mediator between the physical I/O modules, protocol plug-ins, and the application.

2.1.1 phy: Physical network interface

The *phy* interface provides a standard set of APIs for attaching device drivers to VNETI. Those drivers typically deal with networking (mostly RF) devices; however, other I/O devices can also be accessed via VNETI. The interface assumes that information written to/received from the device is packetized.

A *phy* module (device driver) registers itself with VNETI (`tcvphy_reg`), and VNETI assigns it a queue of outgoing packets. Three functions then allow the device driver to (a) retrieve a pointer to the first (topmost) packet in the queue (`tcvphy_top`), (b) extract the first packet from the queue (`tcvphy_get`), and (c) mark a previously-retrieved packet as no longer needed (`tcvphy_end`). Thus, the transmission thread of the driver may be organized into an event loop in which it examines the outgoing queue, e.g.,

```
fsm driver {
  address pkt;
  state LOOP:
    int len;
    if ((pkt = tcvphy_get (0, &len)) == NULL) {
      when (new_packet, LOOP);
    } else {
      start_transmission (pkt, len);
      when (xmit_done, DONE);
    }
    release;
  state DONE:
    tcvphy_end (pkt);
    proceed LOOP;
}
```

When sent over an RF channel, a packet is never physically addressed or encapsulated in any particular way, even if the device implements data-link addressing, handshakes, or any MAC-level features facilitating point-to-point transmission. At the *phy* level, packets are always broadcast and their contents are considered raw, i.e., the entire packet is treated as a sequence of bytes to be made available to the praxis or, more specifically, to VNETI plug-ins.

Packet reception (by a *phy* driver) is implemented by the VNETI function `tcvphy_rcv`. The function presents the newly received packet to the chain of plug-ins which determine the first step of its formal processing.

2.1.2 API: Application interface

A workable VNETI setup involves at least one physical I/O module (phy) and at least one plug-in. Praxis (application) interactions through VNETI deal with *sessions* which are logical entities with the flavor of UNIX file descriptors. At the highest level, a session is an identifier that refers to some specific way of handling packets sent out or received by the praxis, both in terms of the physical interface and the processing by plug-ins. The need for multiple sessions stems from the fact that the praxis may require diverse ways of handling different kinds of packets.

2.1.3 plugs: Plug-in interface

VNETI maintains a single buffer for each outgoing and incoming packet. Rather than introduce layers, it predictably coordinates access to these buffers among registered plug-ins. Each plug-in consists of a data structure containing six function pointers, which VNETI calls at appropriate instances. It calls these entry points when (a) the praxis opens a session that will use the plug-in (`tcv_ope`), (b) the praxis closes a session that used the plug-in (`tcv_clo`), (c) a packet has been received from the praxis for transmission (`tcv_out`), (d) a packet has been transmitted by the physical interface (`tcv_xmt`), (e) a per-packet timer has expired (`tcv_tmt`), and (f) a packet has been received by the physical interface (`tcv_rcv`).

VNETI invokes the latter four packet-centric plug-in functions whenever a packet reaches some stage of processing. These functions provide feedback to VNETI by returning a code that determines the fate of the packet buffer:

`TCV_DSP_XMT` means queue the packet for transmission by the physical module associated with the session.
`TCV_DSP_XMTU` means queue the urgent packet for transmission (at the queue's head).
`TCV_DSP_DROP` means drop the packet and deallocate its buffer.
`TCV_DSP_PASS` means *skip* or *do nothing*, depending on the context.
`TCV_DSP_RCV` means queue the packet for reception at the plug-in's associated session.
`TCV_DSP_RCVU` means queue the urgent packet for reception (at the queue's head).

Each one of these codes is a valid return value for the four packet-centric plug-in functions, resulting in a very flexible framework for handling packets.

To highlight the flexibility, consider the case of an outgoing packet. After a packet arrives from the application, VNETI executes `tcv_out`. Typically, a plug-in will update the buffer and then return `TCV_DSP_XMT`, which causes VNETI to pass the buffer to the transceiver. After transmission, VNETI calls `tcv_xmt`. A simple plug-in will then request the deletion of the buffer (`TCV_DSP_DROP`). An advanced plug-in, however, could set a per-packet timer and defer further processing until it expires (`TCV_PASS`); after it expires, VNETI calls `tcv_tmt`, which again can determine the fate of the buffer. This advanced case could accommodate plug-in-level acknowledgements.

The reception of packets is similarly flexible. After a packet arrives from the transceiver, VNETI alone cannot identify the appropriate session, so it sequentially invokes `tcv_rcv` for each registered plug-in. Using the previously described codes, a plug-in may (a) claim the packet for its associated session (`TCV_DSP_RCV`) or (b) pass the packet on to the next plug-in (`TCV_DSP_PASS`). The first plug-in whose `tcv_rcv` returns something different from `TCV_DSP_PASS` prevents further scanning, and if no plug-in claims the packet, VNETI drops the packet.

2.2 TARP

The Tiny Ad-hoc Routing Protocol (TARP), implemented within the VNETI framework, illustrates how to implement non-trivial communication schemes that allow peer-to-peer communication as a collaborative distributed task. TARP has been described elsewhere (Olesinski et al., 2003; Gburzynski et al., 2007), but we review its key features here given its relevance to the later case study (Section 3).

Upon receiving a packet, TARP's *default* action is to re-send it. Before falling back to the default, however, it tries find a reason to drop the packet. To this end, it uses a set (or chain) of rules that a node applies to the contents of received packets. If a rule matches a packet, TARP drops the packet; otherwise, a lack of knowledge tends to translate into overly altruistic collaboration.

2.2.1 The essential rules of TARP

The first two rules of TARP restrict the extent of the flooding in rather straightforward ways. The first limits the number of hops that a single packet can travel. Meanwhile, the second rule named *DD* (for Duplicate Discard), drops packets with a *signature* matching an entry within a cache of previously forwarded packets.

The third and most powerful rule of TARP is called *SPD* (for Suboptimal Path Discard). Its role is to avoid forwarding in those circumstances when the node believes that its help is not needed, i.e., there are better forwarders already helping the case. The rule uses its own cache to store triplets containing the destination identifier (*N*), the expected number of hops to *N*, and the number of packets dropped by the rule. The node calculates the expected number of hops based on the TARP headers: specifically, headers contain both the number of hops travelled so far and the number of hops last travelled on the reverse path. Given a transmission from *S* to *D*, an intermediate node *K* will drop packets when it determines itself to be too far from the optimal path. The count field addresses changing topologies: after dropping a certain number of packets, nodes try forwarding a packet again in an attempt to discover a new shortest path.

2.2.2 The TARP plug-in

Given an implementation of the requisite caches, the TARP plug-in itself is remarkably simple. Here is the reception function:

```
static int tcv_rcv_tarp (int phy, address pkt,
                         int len, int *ses) {
  if (tarp_ses < 0 || tarp_phy != phy)
    return TCV_DSP_PASS;
  if (hdr(pkt)->snd == my_node_id)
    return TCV_DSP_DROP;
  if (rule_check_dd (pkt))
    return TCV_DSP_DROP;
  if (hdr(pkt)->rcv == my_node_id)
    return TCV_DSP_RCV;
  if (++(hdr(pkt)->hoc) > MAX_HOC)
    return TCV_DSP_DROP;
  if (rule_check_spd (pkt))
    return TCV_DSP_DROP;
  return TCV_DSP_XMT;
}
```

The macro `hdr` provides access to the TARP-specific packet header (casting the packet pointer to a pertinent structure). We see that *DD* is checked before the hop count limit (note that the packet signature for *DD* does not include the hop count). The second `if` statement eliminates packets that have originated at the current node.

The only other not completely trivial function of the plug-in is

```
static int tcv_out_tarp (address pkt) {
  hdr(pkt)->hco = get_hco (hdr(pkt)->dst);
  hdr(pkt)->snd = my_node_id;
  hdr(pkt)->ser = sernum++;
  return TCV_DSP_XMT;
}
```

executed whenever the praxis submits a new outgoing packet. The praxis is responsible for filling in the destination address in the packet header, while this function inserts the backward hop count (extracted from the *SPD* cache), the sender address, and the packet's serial number.

3 A CASE STUDY: THE IL SUPPORT SYSTEM

PicOS, including VNETI and TARP, has been used in several practical setups: a commercial grade asset monitoring system, a sensing network for ecological monitoring (EcoNet), an indoor location tracking system (Haque et al., 2009), and the Smart Condo project (Boers et al., 2009; Stroulia et al., 2009). Recently, we have been experimenting with a pilot version of a campus-wide WSN for non-intrusively monitoring the vital signs of residents in an independent living (IL) facility. This project exposed real-life RF constraints that invalidated our implicit environmental assumptions, thus exhibiting a weak spot of TARP. Fortunately, we were able to fix the problem quite easily, once its nature had become well understood. The requisite modification was performed essentially on-site and on-demand, courtesy of the PicOS/VNETI layer-less holistic structure. The praxis needed no changes at all, while the TARP rules have been extended by a fuzzy variant of the usual acknowledgment mechanism.

The IL network is built around Olsonet's EM-SPCC11 (Olsonet Communications Corporation, 2008) which is a general-purpose wireless mote for prototyping WSN solutions. The network is based on two functionally different types of nodes: (a) *tags* equipped with sensors for monitoring the environment and (b) *pegs* for providing connectivity between the tags and the data collection station (sink). The pegs form an ad hoc network providing mesh connectivity between the tags and the sink, which can be viewed as a semi-infrastructure for the tags.

The peg receiving a report from a tag will send an explicit acknowledgment packet (22 bytes) back to the tag: this one-hop exchange does not involve TARP. The peg will forward the report to the sink in a

44-byte TARP packet. Such a packet is acknowledged by the sink at the praxis level, i.e., in an explicit 12-byte acknowledgment packet sent (over TARP) to the peg. An unacknowledged peg report will be retransmitted roughly at 30-second intervals until overridden by a new report from the tag.

3.1 The problem

While the open-field tests of the IL system yielded satisfactory performance, the deployment at the target site proved disappointing with a highly unpredictable and capricious behaviour of the network. As it turned out, the RF characteristics of the site were particularly malicious.

The deployment of pegs was constrained by the geometry of buildings and the availability of power outlets, forcing them to be laid along corridors where the movement of people and metal equipment would cause drastic disturbances to the propagation of RF signals. The indoor paths would cross with outdoor ones resulting in mixed characteristics of both open-field and bunker-like environments. The same region would exhibit the properties of either environment, on an unpredictable trigger, for periods lasting from seconds to hours. The practical range of a single hop would vary from 20 to 150 meters, translating into occasional periods of "good luck," where a tag was able to reach the sink in a single hop, as opposed to incidents of "bad luck," with 5 hops required to accomplish the same feat.

3.2 The solution

The single most important source of problems with the IL deployment was the whimsically poor quality of a single hop: an accidentally acquired series of longish hops would fool TARP (the *SPD* rule) into rejecting subsequent attempts to forward packets via longer (albeit more reliable) paths. Even though the scheme would recover from the misjudgement after a while, the confusion would have a performance toll, as a few subsequent attempts to use the overly optimistic route would fail.

Within the framework of TARP, there is no concept of a next-hop node, which made explicit acknowledgements impractical. There is, however, a natural way of implementing acknowledgments within the broadcast-based forwarding of TARP. Having retransmitted a packet, the node will wait until it hears its copy retransmitted by another node in the neighbourhood, such that the retransmitted copy is seen to have made more hops than the original. Such an event can be viewed as an implicit acknowledgement that the current node has fulfilled its duty in the sense that the packet has made the hop.

The required modification of TARP consists of adding one more cache where the node will store signatures of packets that have been (re)transmitted, but not yet discarded, awaiting an implicit acknowledgment. Such a packet will be retransmitted a number of times (at some interval) before the protocol gives up, drastically increasing the chances for a successful hop. Given the toolbox of VNETI, the modification turns out to be reasonably straightforward:

```
static int tcv_rcv_tarp (int phy, address pkt,
                                int len, int *ses) {
  rtr_cache_t *re;
  if (tarp_ses < 0 || tarp_phy != phy)
    return TCV_DSP_PASS;
  if ((re = find_rtr (pkt)) != NULL &&
                      re->hoc < hdr(pkt)->hoc)
    rtr_cache_drop (re);
  if (hdr(pkt)->snd == my_node_id)
    return TCV_DSP_DROP;
  if (hdr(pkt)->rcv == my_node_id)
    rtr_ack (pkt);
  if (rule_check_dd (pkt))
    return TCV_DSP_DROP;
  ... /* unchanged from earlier */
  return TCV_DSP_XMT;
}
```

For every received packet, the function first uses its signature to consult a retransmission cache containing hop counts. If the hop count of a received packet copy is larger than the cache value, the function concludes that the retransmitted packet has been acknowledged and drops it from the buffer pool. Otherwise, the function proceeds as before with one exception: it may broadcast a dummy packet (with signature matching the received packet) to implicitly acknowledge a packet making its last hop.

Another modification to the scheme involves the plug-in's transmission function:

```
static int tcv_xmt_tarp (address pkt) {
  rtr_cache_t *re;
  if ((re = find_rtr (pkt)) != NULL) {
    if (++(re->cnt) > MAX_RETRIES) {
      rtr_cache_drop (re);
      return TCV_DSP_DROP;
    }
  } else {
    if (rtr_cache_add (pkt) == ERROR)
      return TCV_DSP_DROP;
  }
  tcvp_settimer (pkt, RETRY_INTERVAL);
  return TCV_DSP_PASS;
}
```

This function is invoked after a packet has been transmitted by the *phy* driver. The function looks up the packet's signature in the retransmission cache. If no

entry is found (the part after `else`), this is the first transmission of the packet – cache it. If a matching entry is found in the cache, the function checks the retransmission counter. If the packet has reached the limit, it is dropped and its signature is removed from the cache. For retained packets, the function sets up a timer for the packet (VNETI operation `tcvp_settimer`) and returns `TCV_DSP_PASS`.

When the timer goes off, VNETI calls the plug-in's timer expiration function

```
static int tcv_tmt_tarp (address pkt) {
  return TCV_DSP_XMTU;
}
```

which simply sends the packet back to the transmit queue of the *phy*.

After adopting the solution sketched above, the performance of our IL network improved dramatically. The pilot system has been in use for six months, now passing all tests with flying colours.

4 CONCLUSIONS

We have discussed a layer-less, yet structured, system for developing applications for WSNs built using devices with few resources. The philosophy embraced in our work is to serve the needs of the application first, and the network second. Attempting to harness the channel to appear as a *reliable*, equivalent to "wired", connection has been the root of considerable complexity to wireless networking that has yet to bring any workable solution for small footprint devices. Instead of employing complexity to produce "virtual wires," we argue that we should be enhancing the collaboration of nodes in a communal effort to produce the desired application outcome.

Looking at the IL praxis discussed in Section 3, we see that some communication scenarios (tag-to-peg) intrinsically involve a single hop, while some others (peg-to-sink) call for a special case of network-layer forwarding (many-to-one and back). TARP, with its parameterizable and malleable rules, and the architecture of VNETI, appear sufficient to cater to the application demands.

REFERENCES

Abrach, H., Bhatti, S., Carlson, J., Dai, H., Rose, J., Sheth, A., Shucker, B., Deng, J., and Han, R. (2003). MANTIS: system support for MultimodAl NeTworks of In-situ Sensors. In *WSNA '03: Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications*, pages 50–59, New York, NY. ACM.

Akhmetshina, E., Gburzynski, P., and Vizeacoumar, F. (2003). PicOS: A tiny operating system for extremely small embedded platforms. In Arabnia, H. R. and Yang, L. T., editors, *Embedded Systems and Applications*, pages 116–122. CSREA Press.

Beutel, J. (2006). Fast-prototyping using the BTnode platform. In *DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 977–982, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

Boers, N. M., Chodos, D., Huang, J., Stroulia, E., Gburzynski, P., and Nikolaidis, I. (2009). The Smart Condo: Visualizing independent living environments in a virtual world. In *PervasiveHealth '09: Proceedings from the 3rd International Conference on Pervasive Computing Technologies for Healthcare*, London, UK.

Dunkels, A., Gronvall, B., and Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE Intl. Conference on Local Computer Networks*, pages 455–462.

Eswaran, A., Rowe, A., and Rajkumar, R. (2005). Nano-RK: an energy-aware resource-centric RTOS for sensor networks. In *RTSS '05: 26th IEEE Intl. Real-Time Systems Symp.*, pages 256–265, Miami, FL.

Gburzynski, P. (1995). *Protocol Design for Local and Metropolitan Area Networks*. Prentice Hall PTR, Upper Saddle River, NJ.

Gburzynski, P., Kaminska, B., and Olesinski, W. (2007). A tiny and efficient wireless ad-hoc protocol for low-cost sensor networks. In *DATE '07: Proc. of the Conference on Design, Automation and Test in Europe*, pages 1557–1562, San Jose, CA. EDA Consortium.

Gburzynski, P. and Nikolaidis, I. (2006). Wireless network simulation extensions in SMURPH/SIDE. In *WSC '06: Proceedings of the 2006 Winter Simulation Conference*, Monterey, California.

Haque, I., Nikolaidis, I., and Gburzynski, P. (2009). A scheme for indoor localization through RF profiling. In *ICC '09: IEEE International Conference on Communications*, Dresden, Germany.

Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. (2005). TinyOS: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148.

Olesinski, W., Rahman, A., and Gburzynski, P. (2003). TARP: A tiny ad-hoc routing protocol for wireless networks. In *ATNAC '03: Proceedings of Australian Telecommunications Networks and Applications Conference*, Melbourne, Australia.

Olsonet Communications Corporation (2008). Platform for R&D in sensor networking. `http://www.olsonet.com/Documents/emspcc11.pdf`.

Stroulia, E., Chodos, D., Boers, N. M., Huang, J., Gburzynski, P., and Nikolaidis, I. (2009). Software engineering for health education and care delivery systems: The Smart Condo project. In *SEHC '09: Proc. from the 31st Intl. Conf. on Software Engineering*, Vancouver, Canada.