PicOS & VNETI: Enabling Real Life Layer-Less WSN Applications

Nicholas M. Boers\textsuperscript{a}, Ioannis Nikolaidis\textsuperscript{a,}\ast, Pawel Gburzynski\textsuperscript{b}, Wlodek Olesinski\textsuperscript{b}

\textsuperscript{a}Department of Computing Science, University of Alberta, 2-21 Athabasca Hall, Edmonton, Alberta T6G 2E8, Canada
\textsuperscript{b}Olsonet Communications Corporation, 51 Wycliffe Street, Ottawa, Ontario K2G 5L9, Canada

Abstract

Networks are traditionally built around a stack of layers, but in the simple devices of a wireless sensor network, the costs associated with this approach become significant. In response, researchers have developed a number of low-footprint operating systems that take non-traditional approaches to network abstractions, but at the same time, still 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 (API), (b) protocol plug-ins, and (c) a physical input/output (I/O) module, allows for an intuitive 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.

Keywords: wireless sensor network, operating system, programming interfaces

1. Introduction

Networks are traditionally built around a stack of layers, where each layer provides a set of services via a clearly defined interface \cite{1}. By isolating each layer and assigning it clear responsibilities, this approach allows for an overall reduction in design complexity at the cost of

- increased non-volatile memory usage for storing the additional function implementations,
- increased random access memory (RAM) usage for making function calls between layers and storing local variables at individual layers,
- increased processor usage for traversing the levels of abstraction intervening on the way from general-purpose high-level interfaces down to heterogeneous protocols and devices, and
- increased communication for transmitting data that is functionally duplicated in multiple layers, e.g., checksums.

In general-purpose computing environments, where the machines and operating systems are rather diverse and powerful, the costs are negligible and such layering has proven quite effective. On the other hand, these costs tend to outweigh the benefits in networks consisting of generally homogeneous (networking) hardware with scarce memory and limited processing capabilities.

Our work focuses on wireless sensor networks (WSNs), which consist of low-powered wireless devices commonly called nodes. Even though researchers can deploy them to address a variety of problems \cite{2}, their modus operandi is far from general-purpose computing, and instead, tends to be (a) observing an environment via sensors, (b) possibly affecting it via actuators, and (c) communicating wirelessly with peer devices and/or collection points called sinks. When carrying out these tasks, nodes that are often battery powered use strategies to conserve energy.

The processing requirements at a node are often very low. A node’s primary components include a radio transceiver, microcontroller, energy source, and attached sensors and actuators. Neither sensors nor actuators generally demand much from the microcontroller...
since they often present themselves via simple and direct interfaces. For example, a passive-infrared motion sensor may have a single digital output and a temperature sensor may have an analog output that scales with the temperature. Radio transceivers also use relatively simple interfaces. For some simple transceivers, the microcontroller provides it with data bit-by-bit for immediate wireless broadcast, e.g., the RF Monolithics DM2200 [3]. For more elaborate ones, the transceiver might have a number of registers for storing data prior to transmission and might even check addresses for each reception, e.g., the Texas Instruments CC1100 [4]. In both cases, the hardware interfaces require only a low-powered microcontroller.

The greatest strain on the microcontroller are the functions and capabilities of the software, namely the operating system. Researchers have developed a wide range of operating systems capable of running on these low-power devices, e.g., [5, 6, 7, 8, 9, 10]. Although the most prominent operating system described in the literature is quite possibly TinyOS [6], we focus on a mature and evolving alternative named PicOS [8] which has a number of advantages over the former, most notably:

1. 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). As all threads share the same (global) stack, PicOS’s stack has absolutely no tendency to grow uncontrollably.

2. PicOS is incomparably more flexible with respect to memory. Dynamic memory allocation (even within devices with less than 1 KB of RAM) is its essential feature.

Since 2003, when the first paper on PicOS was published [8], the system has undergone a number of significant changes, and in this work we focus on the latest version. We explore the essentially layer-less design of PicOS and discuss our experience using the presented framework in real-life mesh networks. Section 2 describes the architecture of the operating system and the general syntax of its applications. 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

PicOS is a small-footprint operating system for organizing the multiple activities of an embedded reactive application (*praxis*) executed on a microcontroller with limited resources. It implements a flavor of multitasking within very small amounts of RAM and provides simple, orthogonal, and expressive tools for event-driven I/O and inter-process communication (IPC).

The primary problem with implementing classical multitasking within limited RAM is minimizing the amount of fixed memory resources that must be allocated to every process. The most critical example of such a resource is the stack space: under normal circumstances, every task must receive a separate and contiguous chunk of memory usable as its private stack. Even if the stack size per task is drastically limited, it still remains a significant component of the total amount of memory resources needed to describe and sustain a single task. With the few kilobytes of RAM available on a typical small-footprint microcontroller (e.g., 2 KB on MSP430F148), this problem makes it very difficult to implement classical multitasking involving any non-trivial number of processes.

PicOS solves this problem by adopting a non-classical flavor of multitasking. The different tasks share the same global stack and act as co-routines with multiple entry points and implicit control transfer [11]. 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. Moreover, this representation stimulates clarity – to the extent that blocks of code often appear self-documenting. 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.

PicOS inherits its programming paradigm from SMURPH (also called SIDE) [12], which is a specification and simulation environment for reactive (mostly telecommunication) systems. The fact that PicOS is closely related to SIDE makes it possible to emulate PicOS programs in a realistic virtual environment created in SIDE and named the Virtual Underlay Emulation Engine (VUE2). This SIDE-based emulator allows for flexible modelling of real-life wireless applications.
in a realistic wireless propagation environment [13].

A PicOS praxis, which may be logically split over multiple source files, consists of

1. global data structures and variables,
2. finite state machines (FSMs), and
3. functions that may be called from within the FSMs.

The execution of the praxis begins at the root FSM, which is akin to the main function of a C program. Here is the customary Hello World program for PicOS:

```c
#include "sysio.h"
#include "ser.h"

fsm root {
  state WRITE_MSG:
    ser_out (WRITE_MSG, "hello, world\r\n");
    finish;
}
```

This complete PicOS praxis consists of one FSM, `root`, which contains a single state `WRITE_MSG`. The code within this state writes the message to the device's serial port (UART) using the library function `ser_out`, which has been made available by including the header file `ser.h`. Since a busy serial port will cause this function call to block, its first argument is the state in which it will resume when the port becomes available. If required by the application, PicOS supports a number of ways to interrupt a blocked FSM, e.g., timeouts and inter-process communication (IPC). While one FSM remains blocked, other FSMs can continue to operate independently.

The operating system provides a variety of system calls and library functions, many of which can be selectively included by using a handful of header files. The system calls fall in a number of categories that include

- operations on finite state machines: creating, managing, and destroying FSMs,
- memory allocation: dynamically allocating and freeing memory using the familiar `malloc/free` operations of C,
- power management: selecting the different low-power modes available to the microcontroller for duty cycling,
- inter-process communication (IPC): providing tools for FSM interactions,
- timing: implementing delays for FSMs, including precisely-timed spins and non-busy waiting,
- random number generation: deriving truly random numbers based on external entropy sources,
- debugging: printing debug messages, asserting Boolean expressions, and throwing exceptions, and
- watchdog: providing for the automated resetting of a stalled node.

The operating system also includes libraries for interacting with (a) a selection of abstracted sensors via built-in drivers, (b) the general-purpose input/output (GPIO) pins, (c) light-emitting diodes (LEDs), (d) the universal asynchronous receiver/transmitter (UART), and (e) radio transceivers. In the last two cases, VNETI acts as the single mediation component standardizing the interactions with all those devices (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 a rapid deployment of networked application for microcontrollers, would make it easy to develop testbeds using emulated I/O interfaces. To avoid the protocol layering problems haunting small footprint solutions, the presented interface is essentially layer-less and its semi-complete generic functionality can be redefined by plug-ins. Also, the actual implementation of the physical interface can be encapsulated as a relatively simple and easily exchangeable module. This modularization allows us to easily compile a single PicOS praxis for a number of different hardware platforms. 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.

Figure 2 shows the internal structure of VNETI. In essence, VNETI implements (a) transparent management of buffer (packet) storage organized into a dynamic number of queues with timeouts definable on a per-packet basis, (b) multiple application access points (roughly equivalent to connections or sessions), 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. PicOS’s philosophy of collaboration

Existing communication standards (e.g., ZigBee and Bluetooth) and their proponents promote a view whereby all applications of WSNs require at least a “network layer” as a common prerequisite. Even in academic research dealing with WSNs, the primary focus is on “routing protocols” that strive to solve the communication problem (at the network-layer) once for all. We view this approach as a paradox; it is equivalent to suggesting that in order to form an ad hoc network, one has to first ensure that an underlying protocol provides connectivity between any two peers, A and B. That is, the “enabling” efforts attempt to obsessively ensure a peer-to-peer communication as a prerequisite step to allowing ad hoc communication! Hence, the protocol stack is introduced as an element of those standards, not leaving the freedom to the applications to be expressed over genuinely ad hoc configurations. This assumption about what the applications are supposed to need (without being “asked”) is contrary to the spirit of avoiding layers altogether.

In our opinion, this paradoxical obsession is the most fundamental and debilitating flaw in the industrial approach to mesh networking. It forces upon us complete network-layer paradigms for implementing operations as delicate as routing and forwarding within unknown networks catering to unknown applications. Those paradigms do not fit the system very well. They are based on point-to-point forwarding, whereby a multi-hop path between a pair of communicating end-point nodes consists of a specific sequence of intermediate nodes which must be first identified and then kept track of for as long as the two end-points may want to communicate. Such schemes facilitate connections (i.e., sustained communication sessions involving the same end-points) and assume that paths are relatively stable and reliable. They also go to great pains trying to create an illusion that forwarding is carried out via some isolated semi-wired point-to-point channels directly connecting adjacent intermediate nodes, whereas every transmission in the wireless medium is in fact broadcast. As we have argued elsewhere [14, 15], such schemes are generally incompatible with the small resource and energy footprint of a cheap and sustainable wireless sensor node; besides, they do not even provide the kind of connectivity that would be useful in a typical WSN. This is because WSN applications are characterized by idiosyncratic traffic patterns that mostly do not care about points A and B. A traditional multi-hop path in a typical WSN is a delicate and complicated structure, usually too brittle to be relied on. What the applications need instead is a way to facilitate the distributed collaboration of nodes interconnected via the unreliable and unstable wireless medium. The shaky nature of that medium may occasionally call for altruistic help by nodes not directly involved in a particular case of collaboration. The fundamental problem of dealing with such cases is reducing the impact of poor reliability of the individual helpers, rather than amplifying it by introducing complex and fragile structures as necessary preconditions for the success of a whole (possibly large) collaborative task.

In this context, the role of VNETI, as a module facilitating networked communication among the nodes, is to provide for a convenient way of expressing general communication scenarios in a manner generally devoid of complex prerequisite setups, which we call intrigues. More specifically, by an intrigue we understand a situation where a non-trivial subset of nodes involved in a
collaborative task must first build an elaborate and fragile collaboration mechanism, whose integrity assumes that none of the participating nodes will fail during the entire task. In this sense, a traditional multi-hop path is an intrigue, assuming that the collaborative task in question consists in delivering packets between some (possibly distant) nodes A and B. As we shall see in Section 2.2, such tasks do have solutions devoid of intrigues; unfortunately, the traditional ways of implementing the network layer in WSNs do not amount to such solutions.

Another related issue is the problem of **scalability of effort** of the collaborating nodes. Consider a traditional point-to-point path in a wireless network. The reliability of such a path (e.g., viewed as the packet delivery fraction) is the product of the reliabilities of all its links. This is in fact the quantitative way of expressing the contrivance of the path’s intrigue. But there is another problem: any given node either is a (full) member of the path or it contributes nothing at all. For example, there may be two or three nodes almost equally qualified to act as forwards at a given stage, but the path construction process must decide on only one of them. This is because a point-to-point forwarding scheme pushing packets along the path must know the exact succession (a specific next node) on every step.\(^1\) Considering the capricious and uncertain nature of the wireless medium, 100% reliability of an individual “link” (i.e., hop) in a WSN is a rarity. Consequently, the multiplicative contribution of nodes along a point-to-point path quickly dilutes the reliability of the entire path rendering multi-hop communication highly uncertain (even if bandwidth is not an issue). It would be much better if the multiple nodes could contribute to the path **additively**, i.e., according to their opportunities to help. This is what we call the principle of scalability of effort. It says that distributed algorithms for WSNs should be designed in such a way that an individual node should be able to contribute to the communal task according to its momentary opportunities, i.e., by adding its fractional assistance to the dynamic pool of helpers instead of making the whole enterprise depend on its own well being. The node’s opportunities may be determined, e.g., by the quality of its radio link or the amount of memory available for a cache, and they may be better or worse, depending on the circumstances. For as long as they do exist and are non-trivial, the node’s contribution should always be useful. Needless to say, a traditional point-to-point forwarding scheme, viewed as as a collaborative solution to the problem of communication, does not follow the principle of scalability of effort.

PicOS’s approach to network communication\(^2\) facilitated by VNETI assumes that plug-ins will converse with the system by inspecting packets at various stages of their processing and indicating where they should be directed next. The options include queuing the packet for reception by the praxis, queuing it for transmission by the device, ignoring it (which usually means passing it to the next plug-in), dropping it (meaning discarding and forgetting), or storing the packet awaiting some future action (e.g., to be triggered by a timer). The inspections usually look like rules which constitute a chain, possibly involving multiple plug-ins. In particular, a plug-in may decide that a packet that has arrived from some other node has reached the end of its path. The decision may involve arbitrary conditions based on the packet’s content and need not have anything to do with its formal destination (understood as some transport-layer node address). The packet format need not even provide for an explicit destination address, which may be meaningless for the application at hand.

### 2.1.2. Physical network interface (phy)

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 (notably the UART) can also be accessed via VNETI (as honorary networking devices). The interface assumes that information written to/removed from the device is packetized, in the sense that it is extracted and written in chunks obeying some specific requirements regarding their minimum and maximum size.

A **phy** module (device driver) registers with VNETI by calling this function:

```
int tcvphy_reg (int phy, ofun_t ps, int info);
```

where `phy` is a logical (unique within the praxis) numerical identifier of the module, `ps` is the `options` function (providing hooks for setting and querying some standard options of all drivers), and `info` is a globally unique `information` attribute used to identify the device handled by the driver. The function returns an `event identifier` corresponding to the event that will be triggered whenever the outgoing queue of packets associated with the driver becomes non-empty.

---

1 Schemes that maintain (and sometimes even use in parallel) multiple (but still explicit) paths only multiply the complexity of this problem without solving it: each of the multiple paths is still point-to-point and thus rigid.

2 Strictly speaking, it does not have to be “network” communication: the same VNETI interface is used for I/O with other non-trivial peripherals, e.g., the UART.
A registered phy module is assigned a queue of outgoing packets, which can be accessed using the following three functions:

- `address tcvphy_top (int phy);`
- `address tcvphy_get (int phy, int *len);`
- `void tcvphy_end (address pkt);`

The first one returns a pointer to the first (topmost) packet in the queue (or NULL if the queue is empty), while the second function extracts the first packet from the queue (also returning its length via the second argument). In both cases, the phy argument identifies the phy module (it should be the same as the identifier assigned at the registration (by `tcvphy_reg`). The third function is to be called when the packet, pointed to by the argument, has been transmitted, i.e., it is no longer needed by the driver.

The transmission thread (FSM) of the driver should be organized into an event loop in which it examines the outgoing queue. If the queue is non-empty, the thread should extract and transmit the first packet; otherwise, it should wait for the event whose identifier was returned by `tcvphy_reg`. Here is a skeletal version of such an FSM:

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

Operation `start_transmission` represents the physical action of submitting the packet to the RF device for physical transmission, and `xmit_done` stands for the event triggered by the device when the transmission has been completed (it is typically signalled via an interrupt).

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 this VNETI function:

```c
int tcvphy_rcv (int phy, address p, int len);
```

which accepts the phy number, a buffer with the packet that has been retrieved from the device, and the packet length in bytes. In contrast to the packet pointers commonly handled by VNETI, which typically point to packet buffers that can be put in queues, the second argument of `tcvphy_rcv` points to the raw sequence of bytes that have arrived from the device. The function presents the newly received packet to the chain of plug-ins which determine the first step of its formal processing.

### 2.1.3. The praxis view

A workable VNETI setup involves at least one physical I/O module (phy) and at least one plug-in. Praxis interactions through VNETI deal with sessions which are logical entities with the flavor of file descriptors, e.g., from UNIX. In contrast to a UNIX file descriptor used for networking, i.e., a socket, a VNETI session need not (and usually does not) represent a communication peer. It cannot even be compared to a raw socket, because the actual semantics of a session depend on the plug-ins and can vary drastically depending on their configuration. 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. The need for multiple sessions stems from the fact that the praxis may require diverse ways of handling different kinds of packets. It is conceivable to have different sessions associated with the same networking interface.

The sequence of steps performed by the praxis to set up a session consists of:

1. Initializing the phy. This is usually accomplished by invoking a function associated with the specific device driver, e.g.,

   ```c
   phy_cc1100 (0, 62);
   ```

   which initializes the driver of CC1100 with the phy number 0 and the maximum packet length of 62 bytes. The function carries out the driver-specific initialization invoking `tcvphy_reg` along the way to register the phy with VNETI.
2. Configuring one or more plug-ins. This is accomplished by invoking

```
int tcv_plug (int pl, const tcvplug_t *pl);
```

which takes a logical plug-in ID as the first argument (its role is similar to that of phy ID for tcvphy_reg), and a pack of plug-in functions described by a special structure (see Section 2.1.4).

3. Opening a session by executing

```
int tcv_open (word st, int phy, int pl);
```

The function takes a state number, phy ID, and plug-in ID, and returns a new session ID. The state number is needed in those situations when the function may block, which depends on the plug-in. For example, setting up an elaborate session may involve exchanging packets with other nodes.

Once a session has been created, the praxis can acquire packets from it and/or write packets to it using these operations:

```
address tcv_rnp (word st, int ses);
address tcv_wnp (word st, int ses, int len);
```

The second argument (this applies to both functions) is the session ID. Each session is assigned an input queue where VNETI stores the packets to be received by the praxis from the session, e.g., via tcv_rnp. The operation will block when the queue is empty – hence the (first) state argument: the invoking FSM will be resumed in that state when a packet shows up in the queue. The second function is used to write a packet to the session (i.e., send it out). That works in two stages. First, by calling tcv_rnp the praxis acquires a packet buffer of the specified length (the third argument) associated with the indicated session. The function will block if no memory is available for the buffer, restarting the FSM at the specified state (the first argument) when some memory has been released (freed). Then the praxis can fill in the buffer with the required content. When done, it will call

```
void tcv_endp (address packet);
```

to terminate the packet which, in this case, means that the packet is ready to be actually written to the session. The same function (tcv_endp) called for a packet that has been read with tcv_rnp has the effect of deallocating the packet buffer, which the praxis will do when it has no further use for its contents.

An opened session can be closed by executing tcv_close (s), where s is the session ID. Closing a session deallocates all packets and queues related to it.

2.1.4. Plug-ins

A plug-in is described by a data structure (a pointer to such a structure is accepted by tcv_plug – see Section 2.1.3) containing pointers to six functions. Here is its layout:

```c
typedef struct {
  int (*tcv_ope) (int phy, int ses);
  int (*tcv_clo) (int phy, int ses);
  int (*tcv_rcv) (int phy, address buf, int len, int *ses);
  int (*tcv_out) (address pkt);
  int (*tcv_xmt) (address pkt);
  int (*tcv_tmt) (address pkt);
} tcvplug_t;
```

The first two plug-in functions, tcv_ope and tcv_clo, handle the plug-in-specific actions related to the open and close operations (see tcv_open and tcv_close in Section 2.1.3). Their first argument is the phy ID (as specified to the respective API function) and the second one is the session ID (allocated by tcv_open). The remaining four functions deal with packets, i.e., they are invoked when VNETI needs a plug-in-specific action (or merely a decision) regarding a packet reaching some meaningful stage of processing. Each of those functions returns the so-called disposition code, which can be one of:

- **TCV_DSP_PASS** meaning skip or do nothing, depending on the context.
- **TCV_DSP_DROP** meaning that the packet should be dropped and its buffer deallocated and returned to the free memory pool.
- **TCV_DSP_RCV** meaning that the packet should be queued for reception at the session with which it is associated. A packet may be classified as urgent,\(^3\) in which case it will be queued at the front of the session’s queue; otherwise, it will be queued at the end.
- **TCV_DSP_RCVU** meaning that the packet should be marked as urgent and queued for reception at the session (necessarily at the front of the queue).
- **TCV_DSP_XMT** meaning that the packet should be queued for transmission by the physical

---

\(^3\)This is a buffer attribute used internally by VNETI. Packets as such have no explicit predefined attributes stored in their contents. Thus, the (internal) urgent attribute is never transmitted along with the packet, but is assumed to be derivable from its contents by the plug-in.
module with which it happens to be associated. If the packet is urgent, it will be queued at the front of the module’s outgoing queue (see Section 2.1.2); otherwise, it will be queued at the end.

TCV_DSP_XMTU meaning that the packet should be marked as urgent and then queued for transmission by the respective physical module.

Function tcv_rcv is an exception among the four packet-related plug-in functions in that it operates on raw packet contents (buf) rather than a packet buffer (as the remaining three functions). The function is invoked by tcvphy rcv (Section 2.1.2) to (a) determine whether the plug-in should claim the newly received packet and (b) assign the packet to a session (the session ID is returned via the fourth argument). Value TCV_DSP_PASS returned by tcv_rcv is interpreted as an indication that the present plug-in does not want to claim the packet, i.e., the packet does not fall under its jurisdiction. Another way to express that would be to say the packet does not belong to the protocol handled by the plug-in, except that we prefer to avoid drawing unnecessary boundaries or assigning things rigidly to non-existent drawers. Note that the function makes this decision based on the packet’s entire content (there are no predefined fields in the packet whose meaning would be universal). Consequently, the interpretation of such a decision may be more general. For example, multiple plug-ins may implement the same protocol (whatever that means) and operate as a chain of rules whose purpose is to diversify the processing of packets depending on some differences in their contents.

If the praxis defines multiple plug-ins associated with the same phy (see function tcv plug in Section 2.1.3), all of them are scanned upon the reception of every packet via the phy, and the first plug-in whose tcv_rcv returns something different from TCV_DSP_PASS claims the packet, i.e., the scanning stops there. For the purpose of this scanning (which is the only situation when that matters), the plug-ins are assumed to be arranged in the reverse order of their association by tcv plug, i.e., the last-associated plug-in is scanned first. Thus, the first plug-in can be viewed as a fall-back (or default) plug-in. If no plug-in claims the packet (all instances of tcv_rcv return TCV_DSP_PASS), the packet is dropped. The effect is the same as if one of those functions returned TCV_DSP_DROP.

Any buffered packet, i.e., any packet handled by VNETI following the reception/acceptance stage represented by tcv_rcv, is always assigned to a specific session. For a packet received from the network, this assignment is done by tcv_rcv. For a packet issued by the praxis (e.g., with tcv_snpu – see Section 2.1.3), the assignment is clear from the very beginning.

Note that tcv_rcv has no obligation to assign the packet to the reception queue of the session (which would happen if the function returned TCV_DSP_RCVR or TCV_DSP_RCVU). In particular, it may decide to drop the packet altogether or even direct it to the transmission queue (by returning TCV_DSP_XMT or TCV_DSP_XMTU), in which case the received packet will be immediately queued for retransmission by the node without ever having been seen by the praxis. The plug-in functions have access to several operations of VNETI, which allow them to create new packets, assign them to sessions, clone packets, and so on. Thus, it is easy, e.g., to pass a received packet to the praxis while simultaneously queueing its (possibly modified) copy for retransmission.

Regarding the remaining plug-in functions, tcv_out is called for a packet that has been just submitted by the praxis, typically by tcv_snpu (or rather by tcv_outp following tcv_snpu – Section 2.1.3). Function tcv_xmt is invoked for a packet that has just been transmitted by the phy, which moment is marked by the phy driver calling tcvphy_end (see Section 2.1.2).

A plug-in function may set up a timer for any packet that has been stored in one of VNETI’s buffers. This is in addition to the packet’s normal path through the system, which means that a packet may be present in one of the standard queues (e.g., in the phy transmission queue waiting for transmission) while also waiting for its timer to go off. When the timer goes off, tcv_xmt will be called for the packet. As with the other packet-related plug-in functions, the value returned by tcv_xmt indicates the packet’s subsequent fate.

2.1.5. The null plug-in

VNETI cannot function without a plug-in: one is required even for the simplest conceivable network functionality, like sending packets into the neighbourhood and receiving all overheard packets. This kind of functionality is described by the null plug-in whose simple structure well illustrates the meaning of all the plug-in functions. The plug-in description starts with the list of the function headers and the definition of the plug-in structure:

```c
static int
tcv_ope_null (int, int),
```

---

4There is an urgent variant of the function called tcv_snpu.
The null plug-in does not need the timeout function (*tcv_tmt* – see Section 2.1.4) because it never sets timers for packets, so the respective pointer in the plug-in structure can be set to NULL. Here is the plug-in-specific open action:

```c
static int null_ses = -1, null_phy = -1;
static int tcv_ope_null (int phy, int ses) {
    if (null_ses >= 0)
        return ERROR;
    null_ses = ses;
    null_phy = phy;
    return 0;
}
```

Most of the actual work is done by VNETI. The plug-in function only confirms that the session does not already exist (the plug-in assumes at most one null session per praxis) and stores the session parameters, i.e., the session ID assigned by VNETI and the *phy* ID associated with the session. Value 0 returned by the function means that the plug-in-specific part of the open action has succeeded.

We shall skip the trivial close function. In fact, all functions of the null plug-in are extremely simple: their main purpose is to set the disposition codes needed by VNETI to know what to do with packets, like in these two functions:

```c
static int tcv_out_null (address pkt) {
    return TCV_DSP_XMT;
}
static int tcv_xmt_null (address pkt) {
    return TCV_DSP_DROP;
}
```

which say that (a) any packet written to the session by the praxis (e.g., by *tcv_wnp* – Section 2.1.3) should be immediately queued for transmission at the session’s *phy* and (b) as soon as a packet has been transmitted by the *phy* driver, it should be dropped, i.e., removed and deallocated.

The reception function is a bit less trivial:

```c
static int tcv_rcv_null (int phy, address pkt, int len, int *ses) {
    if (null_ses < 0 || null_phy != phy)
        return TCV_DSP_PASS;
    else
        return TCV_DSP_RCV;
}
```

If the session is not open or the packet arrives on a different *phy* (than the one associated with the session), the plug-in does not claim the packet (the function returns TCV_DSP_PASS). Otherwise, the function returns TCV_DSP_RCV to indicate the the packet should be stored for reception by the praxis at the end of the session’s receive queue.

### 2.2. TARP

The basic and generic idea of TARP, the Tiny Ad-hoc Routing Protocol, has been described elsewhere [16, 14]. Here we shall focus on some implementation aspects highlighting TARP’s role as a layer-less facilitator of the kind of node collaboration hinted at in Section 2.1.1. The role of TARP is to implement peer-to-peer communication in a wireless ad hoc network as a collaborative distributed task in a way that eliminates intrigues and provides for a good scalability of effort.

#### 2.2.1. Rule based forwarding

Consider a generic forwarding scheme operating according to the following principle. A node willing to send a packet to some destination simply sends it out, i.e., broadcasts it to all neighbours. This is how a *phy* module in VNETI transmits packets (see Section 2.1.2). A node receiving a packet checks its destination address. If the node happens to be the intended recipient, the packet has reached the end of its path. Otherwise, the node *may* decide to re-send the packet. This scheme is generic because, as we will see, its essence is in the exact meaning of the word “may.”

Note that all forwarding protocols for wireless ad hoc networks, including the point-to-point ones, implement (mostly inadvertently) a variant of this scheme. This is because every transmitted packet is in fact broadcast: in principle, it can be received (and creatively processed) by all nodes within the transmitter’s range.

In its utmost simplicity, the scheme provides a framework for an infinite collection of communication schemes applicable to the kind of distributed system represented by an ad hoc wireless (sensor) network. Observe that the scheme need not concern itself with the notion of a formal “destination” unnecessarily carried over from the traditional (and restrictive) paradigm of transport/network-layer communication. In the most general terms, when a node receives a packet (from any of its in-range neighbours), it can:
1. use the packet’s content for whatever purpose it sees fit,

2. re-send the packet, such that its neighbours can receive it as well

Needless to say, the node can creatively modify the packet before retransmission. Moreover, there is no need to talk about an explicit source or destination of such a packet. While traditional transport-layer communication schemes, including peer-to-peer exchange and (selective) multicast, can be viewed as special-case effects of a forwarding scheme based on the above approach, they are not required as prerequisites for effective network communication. Consider, for example, what happens with point-to-point forwarding. The only situation when a node receiving a packet from the network decides to retransmit it occurs when (a) the packet has been explicitly addressed to the node as the next step along the path and (b) the node knows the identity of the next node, i.e., it is aware of the route to the destination. Thus, in order to assist the network in the task of conveying information from point A to point B, a node must be informed about its very specific role in the collaborative setup. The objective of a routing scheme in such a system is to keep the nodes up to date about the exact circumstances when they should retransmit received packets (and how a retransmitting node should tag the retransmitted packet with the identity of its next forwarder). A minute flaw in this information, i.e., a single broken link in a path, translates into a broken path, i.e., a failure of the entire intrigue.

With TARP, we reverse this paradigm by postulating that retransmission be the default action. The objective of a routing scheme is to feed the nodes with knowledge that will allow them to take exceptions to this default behaviour. Consequently, the lack of knowledge will tend to translate into overly altruistic collaboration. It may result in some wastage of the channel’s bandwidth (when too many nodes unnecessarily retransmit a packet that could have done without that help), but it will not break any elaborate and meticulously established intrigues.

The best way to implement this kind of operation is to devise a set (or chain) of rules that a node receiving a packet will in turn apply to its contents. The objective of a rule is to find a reason why the packet should not be forwarded; we say that the rule succeeds if it finds such a reason. Then, the processing stops (the packet is dropped) and there is no need to consult other rules.

Rules make their decisions based on some information acquired from the network and cached at the node (in the node’s RAM). If there is not enough information to make an informed decision, the rule will fail, i.e., the packet will not be dropped on its account. This way, the node’s limitations (insufficient memory, lack of opportunities to receive/overhear information due to poor connectivity, etc.) will translate into overenthusiastic support by the node, which can be interpreted as a reduction in the quality of its communal assistance. This way, in addition to avoiding intrigues, the network will be able to fulfil the principle of scalability of effort (Section 2.1.1). This is particularly well illustrated by TARP’s SPD rule (Section 2.2.2).

2.2.2. The essential rules of TARP

Without any rules, the generic forwarding scheme sketched in Section 2.2.1 amounts to unconstrained flooding, which is hardly a workable solution, regardless of the environment. TARP demonstrates how one can introduce rules into the scheme that will constrain the extent of flooding, even down to single-path forwarding (mimicking a traditional point-to-point forwarding scheme), should such a restriction be desirable.\footnote{It practically never is in a serious WSN.}

One of the simplest measures to restrict the extent of flooding is limiting the number of hops that a single packet is allowed to travel (known as the time to live or TTL). This can be viewed as the first obvious rule in TARP’s chain. The second rule is named DD (for Duplicate Discard). The rule compares the signature of a received packet against a list of signatures of recently forwarded packets (stored in a cache). If the signature is found in the cache, the rule succeeds (and the packet is dropped). Note that the size of (the amount of memory allocated to) the DD cache translates into the quality of the rule – in agreement with the principle of scalability of effort.

The third rule of TARP, and the most powerful one, is called SPD (for Suboptimal Path Discard). Its role is to avoid forwarding in those circumstances when the node has grounds to believe that its help is not needed, i.e., there are better forwarders already helping the case.

With TARP, some portion of the packet content is set aside to be used by the protocol (the plug-in) as a header. In addition to the source/destination address pair \(<S, D>\), the TARP-specific components of the packet include: the sequence number of the packet with respect to the sender node \(n\), the number of hops travelled by the packet so far \(h_f\), and the total number of hops travelled by the last packet on the reverse path \(h_b\).
In contrast to the traditional approach to implementing a hop number limit, whereby the remaining hop count of a packet is decremented towards zero, TARP uses an increasing counter incrementing it on each hop towards a globally assumed limit. This is because TARP (the SPD rule) needs to know how many hops the packet has travelled so far.

The rule uses its own cache (the SPD cache) that stores triplets $<N, h_{NK}, C_{NK}>$ indexed by nodes $N$ interpreted as packet destinations, where $K$ is the node caching the triplet (see Figure 3). Whenever $K$ receives a packet sent by node $N$, it extracts $h_f$ from the packet’s header and stores it (as $h_{NK}$) in the (updated) cache entry for $N$. As this operation does not concern duplicates discarded by the DD rule (which is evaluated before SPD), $h_f$ will tend to reflect the current best (shortest) path from $N$ to $K$.

```
Figure 3: The SPD rule.
```

Suppose that $K$ in Figure 3 receives a packet traveling from $S$ to $D$. When $S$ dispatches such a packet, it inserts into its header $h_0$, the total number of hops made by the last packet received from $D$. Again, by the virtue of rule DD, this tends to be the minimum number of hops in which $S$ can be currently reached from $D$. Thus, $K$ evaluates $h_0 - h_f$, which is the expected number of remaining hops to be covered by the packet before it reaches $D$, assuming that it will follow the same (best) path as a previous packet from $S$ that reached $D$ a while ago. If $h_0 - h_f < h_{DK}$, $K$ can suspect that there is a better path from $S$ to $D$ than any path passing through $K$. Thus $K$ may consider its contribution to the forwarding task irrelevant and drop the packet.

This approach has two minor flaws. First, it appears to assume that paths in the $S$ to $D$ direction look the same as the ones from $D$ to $S$, i.e., the symmetry of radio links, which need not always hold in real life [17]. Second, if the rule always strictly follows the inequality $h_0 - h_f < h_{DK}$, i.e., it succeeds whenever the inequality holds, the peers may not be able to recover from node failures or mobility. Therefore, there are two ways to relax the rule. By adding $m > 0$ to the left-hand side of the condition (see Figure 3) the rule will allow paths that appear worse than the best one by a certain margin. This way, the population of nodes involved in sustaining the session between $S$ and $D$ will be larger than strictly needed to follow the shortest paths in the network. Second, each time the rule succeeds for a given destination $N$ (i.e., the packet is dropped), it increments $C_{NK}$. When the counter reaches the threshold $C_{max}$ (which is a parameter of the protocol), the rule forcibly fails. This way, every once in a while, any node perceiving the session at all is given a chance to contribute to the community’s effort of finding the best route.

The above three rules constitute the core of TARP. Note that, in addition to $m$ and $C_{max}$, the scheme is also parameterized by the sizes of the two caches and their eviction policy. The significance of the latter is higher than it may seem at first sight. For example, in applications with a single sink node, one can make sure that the DD and SPD entries pertaining to that node are never evicted from the two caches.

To see how TARP avoid intrigues while following the principle of scalability of effort, consider the situation shown in Figure 4. Packets travelling between $U$ and $V$ are forwarded within the clouded fragment of the network. Suppose that the arrows represent neighbourhoods. In a steady state, the path $A$–$B$–$C$ (of length 2) is the shortest route through the cloud.

```
Figure 4: Additive collaboration in TARP
```

Suppose that the slack parameter $m$ is set to 1. This means that nodes $E$ and $F$ will also retransmit the packets because the route through them (the path $A$–$E$–$F$–$C$) incurs a 1-hop increase over the best path $A$–$B$–$C$. By the virtue of DD, nodes $A$ and $C$ remove the duplicates, which means that, normally, the packets travelling via the longer route will be discarded and ignored at $A$ and $C$. However, if some of those packets do not make it,
e.g., because of the poor channel between $B$ and $C$, the ones arriving via $E$ (at $A$) and $F$ (at $C$) will provide an immediate backup. This way, the efforts of nodes $B$, $E$, $F$ add to the task of providing connectivity between $U$ and $V$, instead of multiplying their percentage contributions.

One of the worst things that can happen in the setup in Figure 4 is the disappearance/failure of node $B$, which is a critical component of the current best path. Note, however, that this disappearance will not disrupt the traffic at all. Now, a would-be duplicate arriving at $A$ or $C$ (from $E$ or $F$), will be bona fide received and forwarded towards the destination. After a short while, as the destinations update their $h_b$ values in response to the increased number of hops along the best path, the nodes within the cloud will learn that $A-E-F-C$ is the best path at the time. Then, with $m$ equal 1, nodes $D$ and $G$ (located on the second best path) will also become involved in forwarding, thus providing immediate backup in case of subsequent mishaps.

TARP is open for modifications and enhancements by adding more rules and/or modifying the behaviour of the existing ones – beyond the simple parameterization. For example, one redundancy problem that SPD is unable to solve is caused by possible multiple paths with the same smallest number of hops. Consider the situation depicted in Figure 5. Even with the most restrictive setting of the slack parameter, $m = 0$, both paths $<K_1, K_2, K_3>$ and $<L_1, L_2, L_3>$ will be occupied by the packets travelling between $S$ and $D$. The duplicates will be eliminated at $A$ (for the $D-S$ direction) and $B$ (for the direction from $S$ to $D$); however, each of the $K$ and $L$ nodes will be consistently forwarding them because, according to SPD, each of those nodes is located on the shortest path between $S$ and $D$. The problem is usually not serious (to the point of being ignored in all present real-life deployments of TARP), but it may become nasty in dense networks, especially if the two (generally multiple) rows of nodes can hear each other. This is because the redundant traffic will then contribute to the noise in the neighbourhood and feed into the congestion.

The situation can be remedied by adding a Boolean flag, labelled $opf$ to the TARP packet header, to be set by a forwarding node when it knows that the packet is being forwarded on one of the best paths, i.e., the SPD rule fails non-forcibly. This means that the packet should normally reach the destination, unless some nodes have moved away or failed. Consider nodes $K_1$ and $L_1$ in Figure 5 receiving a packet from node $A$. Owing to the collision avoidance mechanism involving randomized retransmission delays, one of these nodes, say $K_1$ will be first to re-broadcast the packet. The other node, $L_1$ will yield to this transmission and overhear (receive) the packet re-broadcast by $K_1$. Normally, that packet would be diagnosed as a duplicate and promptly discarded by $DD$. However, if $opf$ is set in the packet header, $DD$ yields to another rule, which compares the signature of the received packet against the signatures of all packets currently queued for transmission. If a matching packet is found at $L_1$ and its $h_f$ is not less than $h_f - 1$ in the received duplicate, then the packet at $L_1$ is dropped. In plain words, this means that by forwarding its copy of the packet, $L_1$ would not improve upon the forwarding opportunities already extended by $K_1$. This mechanism will not help if the paths are disjoint, but it will kick in wherever they cross. Note that while long disjoint paths of the same length need not be rare in a realistic network, the ones for which the length is the shortest possible definitely are.

One reason why we mention this possible way of augmenting the canonical set of TARP rules is its illustration of a cross-layer approach. There is no problem implementing this kind of operation (rule) in TARP, because of the layer-less (holistic) implementation of VNETI, with TARP functioning as its plug-in. Note, however, that the action of comparing a “network-layer” packet signature to the signatures of all packets queued for “data-link-layer” transmission is a vivid violation of traditional rules of layering. Tricks like this have proved extremely useful (and natural to play within VNETI’s plug-in environment) for adding/augmenting various rules of our collaboration schemes.

2.2.3. The TARP plug-in

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

---

Footnote: PicOS phy drivers implement such mechanisms based on listening before transmission (LBT). Many commercial RF modules, e.g., CC1100, provide built-in tools facilitating such simple (handshake-free) collision avoidance schemes.
reception function:

```c
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 (thdr(pkt)->snd == my_node_id)
        return TCV_DSP_DROP;
    if (rule_check_dd (pkt))
        return TCV_DSP_DROP;
    if (thdr(pkt)->rcv == my_node_id)
        return TCV_DSP_RCV;
    if (++(thdr(pkt)->hoc) > MAX_HOC)
        return TCV_DSP_DROP;
    if (rule_check_spd (pkt))
        return TCV_DSP_DROP;
    return TCV_DSP_XMT;
}
```

The macro `thdr` 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 been sent (originated) by the current node. If that `if` succeeds, it means that we are looking at a packet that has looped back to its sender, which is clearly a perfect candidate for dropping.

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

```c
static int tcv_out_tarp (address pkt) {
    thdr(pkt)->hco = get_hco (thdr(pkt)->dst);
    thdr(pkt)->snd = my_node_id;
    thdr(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) and the sender address. If there is no information regarding the backward hop count from the packet’s destination, `get_hco` will return a maximum value representing infinity. The packet’s serial number is global per sender (it could be calculated on a per-destination basis which, however, could pose non-trivial memory requirements). Thus, the packet’s signature for `DD` comprises the sender ID and the serial number.

The real set of TARP functions used in a production system is a bit more complicated, owing to the need to accommodate broadcast traffic as well as some special (e.g., diagnostic) messages that have to receive special treatment. The complication mostly consists in recognizing special message types and subjecting them to different chains of rules.

3. A case study: the IL support system

PicOS, including VNETI and TARP, has been used in several practical setups, including a commercial grade asset monitoring system, EcoNet – a sensing network for ecological monitoring, an indoor location tracking system [18], and the Smart Condo project [19, 20]. 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. The last 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 “data-link” layer acknowledgments.\(^7\) This section reports on that experience.

3.1. The system

The IL network is built around Olsonet’s EM-SPCC11 [21] which is a general purpose wireless mote for prototyping WSN solutions. The mote is based on the MSP430F1611 microprocessor by Texas Instruments, which provides a number of digital and analog ports for interfacing sensors, and the CC1100 RF module (also by Texas Instruments), which operates in the 902-928 MHz ISM band at the raw transmission rate of 10 kbps. The microprocessor features 10 KB of RAM, which is more than needed for the application at hand.

The network is based on two functionally different types of nodes: `tags` equipped with sensors for monitoring the environment and `pegs` whose role is to provide connectivity between the tags and the data collection station. The latter is a computer interfaced to a selected `peg` (called the `master`) via a UART (over USB). This kind of setup falls under the umbrella of our generic application blueprint termed `Tags and Pegs`.

The pegs form an ad hoc network providing mesh connectivity between the tags and the master. That network can be viewed as a semi-infrastructure for the

\(^7\)This is the shortest intuitive explanation of the solution. There is no data-link layer in VNETI/TARP.
tags. The latter are battery-powered and expected to operate for months without replacing the batteries; thus, their activities are organized into non-trivial duty cycles. Pegs, on the other hand, are connected to wall outlets, and energy consumption is not an issue for them.

Tags are equipped with two types of sensors: IR motion detectors (whose purpose is to detect human movement within a room) and light sensors (indicating whether the lights in that room are on or off). A tag wakes up every two seconds and reads the indications of its two sensors. Those indications are aggregated and reported at 15-minute intervals to the network, i.e., the nearest peg. Along with the aggregated readings of its two sensors, the tag reports its battery status (which can be viewed as one more sensor reading). The size of a report packet is 32 bytes.

The peg receiving the report will send an explicit acknowledgment packet (22 bytes long) back to the tag. Note that this exchange is always one-hop and does not involve TARP. Having sent its report, the tag will keep the RF channel open (its receiver on) for up to 2 seconds (expecting an acknowledgment). It will retry the report up to 3 times if the acknowledgment does not arrive.

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

One more packet type is the 16-byte master beacon broadcast by the master peg roughly at 30-second intervals.

3.2. 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 unacceptable) delivery rate and a 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 (most notably) 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. That would translate into occasional periods of “good luck,” where a tag was able to reach the master node in a single hop, as opposed to unpredictable incidents of “bad luck,” with 5 hops required to accomplish the same feat.

Notably, our previous experience with TARP in practical deployments had taught us to cope with some of the deficiencies of real-life surroundings. With one such technique, dubbed route recovery (or RTE), the slack parameter \( m \) (Section 2.2.2) would be incremented after some number of packet drops by the SPD rule. Note, however, that increasing \( m \) to smooth out occasional lapses in local communication only makes sense in reasonably uniform and balanced environments. While an increased \( m \) will formally compensate for occasional hiccups along the “best” path, doing so indiscriminately will effectively transform the scheme into flooding. This is particularly true in unbalanced deployments, e.g., with narrow corridors, where even a moderate nonzero value of \( m \) will tend to engage all nodes in a given area.

Another technique that proved useful for avoiding substandard channels in some environments was to avoid updating the SPD cache when the signal level of the received packet (RSS) was below a configured threshold. That way, the SPD rule would not jump to conclusions about attractively short paths resulting from accidentally acquired hops. Again, such a simple trick is not much help when the RF properties of the environment tend to fluctuate largely and unpredictably. In our case, two packets received over the same distance would often exhibit drastically different RSS levels.

3.3. The solution

If asked to identify the single most important source of problems with the IL deployment, we would say that it was the whimsically poor quality of a single hop. Owing to the fluctuating propagation characteristics of the medium, an accidentally acquired series of longish hops would fool TARP (the SPD rule) into rejecting subsequent attempts to forward packets via longer (albeit more realistic) paths. Even though the scheme would recover from the misjudgement after a while, the confusion would take its toll, as a few subsequent attempts to use the overly optimistic route would fail. The net appearance of that behaviour was a “data-link” failure to convey packets between the neighbours that were supposed to provide a decent connectivity, but then failed to deliver on the promise.

One possible way to ameliorate the problem, naturally applicable in the layered word, would be to retransmit the data-link packet several times expecting an
acknowledgment from the next-hop node. Even if the previous hop was a lucky fluke (and the opportunities have changed) it is unlikely that the route is now completely unusable. Giving up after a single attempt appears too pessimistic in this context: perhaps the packet will make it after all, given a few more tries.

Within the framework of TARP, there is no concept of a next-hop node: the packet is received by all neighbours that can hear it; then, some of those nodes may decide to retransmit the packet. Consequently, it is not clear which of them should acknowledge the received packet. One obvious observation is that a node whose rules prescribe no retransmission should not do it for sure. This observation also hints at 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 indication that the current node has fulfilled its duty in the sense that the packet has made the hop, i.e., has been successfully passed to another node which subsequently has considered itself relevant for the forwarding task.

The required modification of TARP consists in 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. For every received packet, the function first consults the retransmission cache by trying to look up the packet’s signature (determined by the pair \(<\text{sender ID}, \text{serial number}>\)). One item stored in the cache is the original hop count of the retransmitted packet. If a matching entry is found in the cache and its hop count is less than the hop count of the received copy, 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: if the received packet happens to be addressed to this node, the plug-in forces the transmission of a dummy packet whose signature matches the received packet with the properly updated hop count (this is performed by \texttt{rtr\_ack}). This is required to acknowledge a packet making its last hop: such a packet will not be forwarded any further, so its acknowledgment must be explicit.

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

```c
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;
}
```

invoked whenever a packet has been transmitted by the phy driver. Again, the function looks up the packet’s signature in the retransmission cache. If no entry is found (the part after else), it means that this is the first transmission of the packet, in which case a pertinent entry is added to the cache. The operation, carried out by \texttt{rtr\_cache\_add}, will fail if there is no more room in the cache, in which case the packet is simply dropped – exactly as in the original version of the scheme.9 Otherwise, the retransmission count of the

---

8Note that internal nodes (i.e., non-master pegs), which never originate or absorb packets, need not formally have network addresses.

9Note that the new mechanism obeys the principle of scalability of
new entry (re->cnt) is set to 1.

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. Otherwise, the function sets up a timer for the packet (VNETI operation tcp_settimer) and returns TCV_DSP_PASS, which means that the packet will be kept in the buffer pool, waiting for the timer to go off, without being queued for any other kind of processing. When the timer goes off, VNETI will call this function of the plug-in:

```c
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 drastically. The pilot system has been in use for two months, now passing all tests with flying colours.

4. Conclusions

We have discussed a layer-less, yet highly structured, system for developing applications for wireless ad hoc networks built of tiny devices with minimalistic resources. We believe that the key to a successful organization of activities in such a network is to reject the atavistic and instinctive pressure to see the network first and the application second. It is not enough to remove all the abominable layers from networking: the networking itself must not be seen as a layer that comes with a predefined set of communication primitives. Instead, it must provide and encourage flexible ways of organizing collaborative efforts within a distributed population of nodes interconnected through a flaky channel. Traditional views on harnessing that channel for reliable communication have brought considerable complexity and diversity to wireless networking but, as far as we can tell, have failed to bring the kind of workable and maladaptive rules, is able to cater quite well to the demands of such forwarding, but even it is an overkill, if postulated as a general scheme to be good for all occasions. The problem that a wireless network may want to solve need not call for complete connectivity among all nodes (even the IL praxis does not need that much), and the application-constrained traffic patterns may be best tackled by a completely different set of rules than those of TARP.

For illustration, suppose that the role of a hypothetical WSN is to calculate a reasonable estimate of the average of the numerical readings of all its sensors. Such a network can operate by circulating controllably superfluous packets where nodes register their values (in an apparently chaotic and accidental manner) until the sink receives a packet in which a significant fraction of all the nodes have registered. There is no implied or prerequisite need for reaching specific end-nodes with packets that would be specifically addressed just to them. This scenario may look wasteful at first sight, but so was TARP after its introduction in Section 2.2.1, before the presentation of its rules in Section 2.2.2. So it is all in those rules that creatively conspire to eliminate most of the unnecessary re-broadcasts from the scene. While routing can be viewed as a special case of a distributed computing problem, many other problems can be solved in the way of TARP without assuming routing as a prerequisite.

References


