A WSN ARCHITECTURE FOR BUILDING RESILIENT, REACTIVE AND SECURE WIRELESS SENSING SYSTEMS

PAWEŁ GBURZYŃSKI[1,2,3]

[1] *Faculty of Art, Technology and Communication,*
Vistula University,
ul. Stokłosy 3, 02-787 Warsaw, Poland

[2] *Professor Emeritus, Department of Computing Science,*
University of Alberta,
Edmonton, Alberta, Canada T6G 2E8

[3] *CTO, Olsonet Communications,*
109 St. Claire Avenue, Ottawa, Ontario, Canada K2G 2A7

(received: 17 June 2021; revised: 12 August 2021;
accepted: 13 August 2021; published online: 30 October 2021)

Abstract: We introduce a wireless sensor network (WSN) architecture intended for massive deployments in custom applications where the primary goal is the collection of low-volume (e.g., telemetric) data possibly augmented with spontaneous special events, like alerts or alarms. The network is built of inexpensive, small-footprint, energy-frugal, possibly mobile nodes running reactive programs and self-organizing themselves into resilient distributed systems in a manner embracing the limited capabilities of the devices as well as the unreliable nature of ad-hoc wireless communication. We propose and elaborate on a holistic approach to constructing complete WSN applications. Our approach incorporates a certain unified p rogramming and communication paradigm. In addition to producing small, energy-efficient, self-documenting and reliable programs for ultra-small-footprint motes, that paradigm enables authoritative virtual execution of complete application, thus facilitating their rapid development, testing, augmentation and modification.

Keywords: Wireless Sensor Networks, Mesh Networks, Reliability, Security, Internet of Things

DOI: https://doi.org/10.17466/tq2021/25.2/b

1. Introduction

An ad-hoc wireless network is a distributed system where (typically) simple and often inherently mobile (or easily movable) processing devices (nodes) inter-operate in a spontaneous manner that does not depend on a pre-existing communication infrastructure. In a sensing system organized after this fashion, i.e., in a wireless sensor network (WSN), the devices (at least some of them) are equipped with sensors and/or actuators interfacing them to the environment. The network's purpose is then to collect data pertaining to some distributed process and (possibly) affect that process via the actuators. All the WSNs that we are concerned with in this paper admit (at least in principle) both types of interfaces, i.e., sensors as well as actuators.[1]

Node activities may involve non-trivial processing of the collected data, e.g., aggregation or an abstraction of the sensing/actuating hardware into virtual sensors/actuators [2, 3]. Typically the network is interfaced to an external system which we shall refer to as OSS (Operational Support System) executed on a server or a workstation. The interface is provided by a selected node dubbed the master node. Sometimes, e.g., for improved reliability, the master node may be replicated into several copies possibly spread geographically over the WSN area.

Regardless of the nature and complexity of the processing carried out within the network nodes, the character of that processing is typically strongly reactive. Using the operating systems terminology we would say that the nodes are I/O-bound, i.e., they spend most of their time waiting for events [4]. If there is a limit to the processing power of a node, it stems from the limited ability to receive events, as opposed to the CPU power needed to handle them. Any number crunching components in the processing of the data received from the sensors and/or passed to the actuators are implemented in the OSS.[2]

One important feature of the ad-hoc (mesh) networking paradigm that we want to promulgate in this paper is multi-hop forwarding [6]. These days one often gets the impression that ad-hoc networking and multi-hopping have been rendered obsolete by the IoT bandwagon [7]. It is certainly true that many practical cases of wireless sensing can be handled effectively with a direct single-hop connectivity of the sensor node to an Internet access point, e.g., over Low-Energy Bluetooth (BT LE) [8], ZigBee (IEEE 802.15.4) [9], or other schemes [10, 11]. Nonetheless, there exist important niches where mesh networking, as seen from the angle of WSNs, appears attractive. Mission critical networks [12, 13], industrial control networks [14, 15], ecological monitoring/preventive sensing networks [16–18], disaster response and recovery systems [19], military networks [20, 21], custom security systems [22, 23], are the few examples that come to mind. The reasons why a system may prefer not to rely on the infrastructure for all communication can be stressed in these points:

1. The infrastructure may not be available. This concerns potentially massive networks deployed in areas with limited coverage, e.g., ecological monitoring systems [16] or systems deployed in forests for fire prevention and detection [17, 18].

[1] Some authors [1] use the acronym WSAN (Wireless Sensor and Actuator Networks) for such systems.

[2] One example of such a processing component is location tracking service [5].

2. The massive nature of the network naturally calls for local (internal) communication among the nodes instead of using the infrastructure as a conceptually single access point.

T he functionality of the network, viewed as a distributed processing system, requires a neighborhood concept (local communication) and a creative interpretation of that concept.

3. The proprietary nature of the system and/or security/reliability issues preclude dependence on any external requisite service.

From our point of view, the most serious problem with infrastructure-based networks is the rather drastic reduction of functionality imposed by the sensor-gateway hierarchy. While some contemporary RF communication technologies pushed with the IoT bandwagon allow in principle for long-range, low-power (LPWAN) communication between sensing devices and a small set of infrastructure access points [24, 25], the implied architecture of such a WSN is strongly geared towards low-bandwidth passive data collection precluding the kind of spontaneous communication that we find extremely useful for implementing various kinds of virtual sensors [3]. If all data exchanges in the network have to be mediated through fixed and sparse gateways, according to rigid and limiting schedules [26], then the WSN (viewed as a distributed processing system) is bound to lose a significant fraction of its potential. This purely functional argument comes on top of the concerns one may have about outsourcing communication for a mission-critical/proprietary system.

2. The hardware

A sensing node consists of a microcontroller (MCU), a radio (RF) module and a number of sensors and/or actuators interfaced to the MCU. We target, or at least are prepared to accommodate, the lowest end of the microcontroller spectrum. This is to say that our software is able to run in devices with as little as 256 bytes of RAM (for data) and 4KB of flash memory (for code). For radio communication, our networks operate in the sub-1GHz area of the ISM (Industrial, Scientific, Medical) band, incorporating frequencies around 433.92 and 915MHz.

2.1. The microcontrollers

The most popular MCU type used as the basis for nodes in our networks has been MSP430 by Texas Instruments [27]. While the 16-bit MSP430 family may be considered aged these days (the MCU was first introduced in 1992), it is hardly obsolete: new implementations of the same logical processor are being conceived to this day [28]. Our new designs incorporate CC1350 (also by Texas Instruments) devised as a new-generation, 32-bit, ARM-based replacement of MSP430, targeting the wide context of the IoT [29]. While MSP430 comes in many flavors, i.e., many different chips with different configurations of RAM, flash, modules and peripherals, there is a single CC1350 variant intended to cater to all the envisioned applications of the MCU through flexible reconfiguration of its vast repertoire of options.

Despite the relatively large time gap separating the two designs, the capabilities of CC1350 are not evidently better than those of some of the MSP430 variants. Table 1 compares the basic parameters of the two types of the processors. Note that there exist MSP430 specimens with more RAM than what is available on CC1350. The primary superiority of CC1350 over the MSP430 family is in its flexibility which makes it possible to replace a large collection of MSP430 variants (forcing the designer to select the one best suited for the application at hand) with a single general-purpose chip.

Table 1. A comparison of the basic parameters of MSP430 and CC1350

Feature	MSP430 (range)	MSP430 (typical)	CC1350
RAM	128 B – 66 KB	4 KB	20 KB + 8 KB cache
Flash (code)	1 KB – 512 KB	40 KB	128 KB
ROM (data)	1 KB – 16 KB	2 KB	None
Clock speed	up to 20 MHz	12 MHz	up to 48 MHz

Our networking solutions pose little demand on the fundamental resources of the MCU, although they can take creative advantage of any spare capacity. Both MCU types provide ample hardware platforms for implementing the functionality of our nodes, including interfacing them to a plethora of sensors and peripheral equipment. Standard industry interfaces, like UART, SPI, I^{2}C, as well as powerful ADC/DAC (Analog to Digital and Digital to Analog Conversion) capabilities are available in both cases.

2.2. The RF modules

The typical standard RF module well exemplifying the functional expectations of our communication schemes is CC1100 by Texas Instruments [30]. A strong factor in favor of using CC1100 in our systems is its integration into the CC430 MCU series. The CC430 prefix refers to the variants of MSP430 with the CC1100 radio incorporated into the chip.

CC1100 offers a number of configuration options. The device is able to transmit and receive unstructured packets with optional automatic address-recognition[3] and integrity-check capabilities. Medium access (collision avoidance) is left to the firmware. The module provides tools for channel status assessment (sensing the channel before transmission) facilitating simple LBT (Listen Before Transmit) schemes [31].

The nominal transmission rate, expressed as the number of bits that can be inserted into the channel per time unit (ignoring framing and interference), is adjustable between hundreds and hundreds of thousands bits per second in a way

[3] Which our communication schemes do not use.

that trades bandwidth for range and incorporates several selectable modulation and FEC (Forward Error Correction) techniques. The effective communication range depends on the environment (including the antennas) and is between a kilometer (low rate in open space) and tens of meters (in RF-unfriendly closed areas). Up to 256 channels are available to separate transmissions for different systems communicating within the same ISM band in the same area.

The maximum length of a single raw packet is in principle unlimited, although practical considerations (bit error rates, buffering capabilities, congestion) precipitate rather drastic restrictions. The maximum length of a single (complete) packet assumed in our networks is 62 bytes. This kind of limitation is perfectly acceptable in a WSN; more serious limits are often imposed [32].

Devised as a Swiss-Army knife for IoT applications, CC1350 comes with an integrated RF module [29] being partially compatible with CC1100 (making inter-operation possible), but providing more configuration options, including BT (also BT LE), ZigBee and the so-called proprietary mode which basically means raw packets parameterized in a way akin to CC1100. From our perspective, the features of both types of modules are very similar. One of them is the availability of RSSI (Received Signal Strength Indication). RSSI can be treated as a sensor, e.g., enabling location-based services [5].

2.3. The sensors

A node can accommodate many diverse types of sensors and actuators whose assortment is application-specific. They range from analog devices, interfaced via ADC/DAC, to digital sensors and actuators connected via SPI or I^{2}C, including IMU,[4] humidity, light, pressure, buzzers, door locks, various switches, motor controllers and so on.

A typical sensor delivers some data upon request (when polled by the program) and may also trigger events (implemented as interrupts to the MCU). We assume that the amount of data delivered by a sensor is never overwhelming, in the sense that it does not strain the computing powers of the node and does not cause unrealistically heavy traffic in the network. A node need not act as a dumb pusher of whatever data it retrieves from the sensor: the data may be processed locally, or perhaps in collaboration with the neighbors, before its digest is expedited towards the OSS. Generally, whatever happens to the data, we assume that the nodes do not get involved in heavy computations and the nature of their activities is reactive. This is important from the viewpoint of the node's power budget (Section 2.4) as well as its emulation (Section 3.2.2).

Some applications may pose tradeoffs in this respect. For example, a single full readout from an IMU sensor may amount to 9 values (three coordinates per each of the three basic functions) which may translate into 18 bytes (16 bits per value). Extracting this many bytes from the sensor and expediting them over the

[4] Inertial Measurement Unit. Such a sensor amounts to an accelerometer, usually augmented with a gyro and a compass.

RF channel does not look like a serious problem; however, the application may be interested in detecting complex patterns in the indications of the sensor taken continuously at a nontrivial rate (say a hundred of samples per second) and over nontrivial time (like minutes or hours). The question is how much processing the node can afford (within the limitations of its resources and power budget) and how much traffic the network can handle. Minimizing the amount of processing carried out at the node (and delegating it to the OSS) may require sending unrealistically large volumes of data over the network. On the other hand, doing all the processing at the node may be unrealistic as well. In such cases, one should aim at a compromise whose exact nature will depend on the application [33, 34].

2.4. Low power operation

Practical WSNs are constrained by two factors: the integral of the cost over all nodes and the energy budget of those nodes that must be self-sufficient for power supply. The self-sufficiency of a node can be achieved with efficient battery-based operation or with energy harvesting techniques [35, 36]. Note, however, that the latter may be incompatible with the low node cost, especially if the network is massive and/or the nodes must be considered disposable, as in some military networks [37] or in disaster response systems [38]. It may happen that the expected longevity of a node renders advanced energy harvesting pointless.

Low power requirements naturally connote small footprint of the node, thus automatically pushing towards the low cost of the hardware. However, owing to the drastic discrepancies in power requirements during the different types of availability of the MCU, enforcing low energy consumption also requires a careful implementation of the firmware and a scrupulous approach to RF communication.

Table 2. Current drain by CC1350 in various CPU states (3V supply)

Mode	**Current**
Shutdown (with power on): the MCU is dormant, but it will wake up and reset on a level change on any of the preconfigured GPIO pins	185 nA
Standby: the lowest-power idle state, basic clocks running, interrupts enabled, CPU halted waiting for an interrupt	1 μA
Active: CPU running	570 μA
Idle: normal idle state with fast transition to running on an interrupt	2 mA
RF receiver on: waiting for reception or receiving a packet	6 mA
RF transmitter on: transmitting a packet	23 mA

Both MCU types of our choice are powered from 3 V sources. Table 2 lists the ballpark current drain by CC1350 in the different states of its operation. The exact actual values may differ slightly, depending on the details,[5] but the differences do not affect the perspective. The values (or their magnitudes) are representative for all microcontrollers in the interesting class, including MSP430.[6] We make two observations:

1. The energy budget is strongly dominated by RF activities. For all practical purposes, the receiver is no less power hungry than the transmitter, especially that transmissions are never physically continuous.
2. When the device has nothing to do, but it has to be ready to respond to an event (practically any event other than a packet reception), it can last for years on any battery.

Suppose that the node is powered from a 1 Ah battery.[7] Staying in a reasonably quickly responsive (standby) state and being able to react to a signal from its peripheral, the device will practically last forever. But in a constant reception mode, when the node persistently waits for a packet from the network, it will run out of energy in about 5 days.

Most sensors (including IMUs and environmental sensors, like temperature, humidity, pressure, light sensors) draw between a few μA and a few mA. Usually, the readouts requiring a larger current can be easily duty cycled, bringing the average current drain per sensor to the level of a few μA. For a movable node equipped with an IMU, it is easy to detect motion based on low-rate sampling of the acceleration and thus discriminate between movement (an active state) and immobility at a very low energy expense.

3. The software

Two main issues make the process of developing programs for WSNs cumbersome. First, programming the resource-constrained tiny devices requires special tools, including operating systems, programming languages and associated utilities, as well as a special approach to programming where the coder is aware of the resource limitations and the idiosyncrasies of the often exotic peripheral equipment. Second, the complete programs are extremely difficult to test before deployment. Note that by a complete program we understand the whole application, i.e., the distributed collection of programs executed by all the nodes as well as the OSS software.

While there exist operating systems and comprehensive programming environments for microcontrollers intended for the kind of reactive applications that we envision for WSNs [39–41] and some of them provide tools facilitating development of distributed wireless applications [42, 43], the problem of authoritative

[5] For example, the current drawn by the transmitter depends on the transmission power setting.

[6] Some minor details differ, e.g., MSP430 has no responsive shutdown state, but its minimum current drain in the lowest-power responsive idle state is ca. 1 μA.

[7] Say, two standard AA-type batteries.

testing of complete systems before their real-world deployment is still an open issue. There is a difference between a realistic MCU emulator, which often allows the developer to test a program to be flashed into a real device in a perfectly virtual world and a realistic emulator of the complete network, possibly comprising thousands of nodes, including the RF channel and the OSS interaction. While efforts have been undertaken in this area [42, 44–48], they all suffer from many limitations, be it oversimplifications of the wireless medium [42, 46], insistence on the accurate emulation of the MCU at the machine instruction level [45], too much abstraction [44, 47], or limitations of scope [48].

3.1. PicOS: the OS and its programming environment

Our programming environment for a network node is defined by PicOS which is a tiny operating system for organizing activities of reactive programs into a flavor of multiple tasks [49–51]. Its objective is to minimize the amount of RAM needed to sustain a single task while making the (thus restricted) multitasking useful. All operating systems targeting MCUs struggle with the same problem trying to solve it in their ways. What makes PicOS stand out in the crowd is the adaptation of a multitasking paradigm that automatically makes the programs eligible for event-driven simulation (emulation) which can be applied to complete (networked) applications [52].

3.1.1. Finite State Machines

The most critical memory resource needed to sustain a task under a traditional operating system is the stack space. Consider an MCU equipped with 2KB of RAM.[8] Whichever way the memory is partitioned, the chunks used as stacks for the multiple tasks steal the precious resource from where it is truly needed. The stack space is practically wasted from the viewpoint of the application, being merely a bureaucratic overhead demanded by the high-level layout of the program, whereas the scarce RAM is needed to accommodate the true data structures that the application actually operates on.

The reason why different tasks need separate stack areas is that a task can be preempted, i.e., its state (including the local variables of its functions) must be preserved while the task is waiting for its turn to run. The reason for the preemption may come from the task itself (the task has to wait for something, e.g., a value to be delivered by a sensor), or from the outside (e.g., a more important task becomes ready to run). In appreciation of this problem, some operating systems for tiny devices, e.g., TinyOS [42, 53] minimize the number of activities that may have to be preempted with their states stashed on separate (fragmented) stacks. The typical approach is to push most of the spontaneous concurrency into interrupt service functions reserving tasks for callbacks, i.e., delayed actions that cannot be carried out immediately from an interrupt service routine. In the original version of TinyOS (the one that was truly geared for tiny MCUs) the

[8] For example, this much RAM is available on MSP430F148 which has been one of the popular choices for our network nodes.

limit on the number of simultaneous tasks was 1. That would solve the problem of multiple fragmented stacks: all interrupt service routines could share the same (global) stack with the single thread. However, a few other problems were brought in. One of them was the rigid (static) memory allocation for the program's data structures. Dynamic memory allocation implies the possibility of blocking and waiting (for the memory to become available) which is difficult to implement if the requesting activity is unable to suspend itself. Another problem was the notoriety of various stack overflow bugs caused by the difficulty in predicting the configurations of interrupt service routines that could be momentarily stacked on top of one another [54, 55].

The solution adopted by PicOS consists in drastically limiting the amount of state information that must be preserved when a task is preempted and eliminating the stack as the place where that information is saved. A similar idea found its way into ConTiki [41, 43] whose authors also realized the bureaucratic and wasteful nature of stacks, especially when seen from the perspective of microcontrollers with a minuscule amount of RAM. The different tasks in PicOS share the same global stack and act as co-routines [56] with multiple entry points and implicit control transfer. A task looks like a Finite State Machine (FSM) that transits through its states in response to events. The CPU is multiplexed among the multiple tasks, but only at state boundaries.

PicOS comes with its programming language which looks like C augmented with a few additional keywords and constructs. Owing to its close integration with the system, the language has no specific name: we just call the whole thing PicOS identifying the language with the system. A program in PicOS is preprocessed into C by a tool named PiComp and then compiled by the standard (GNU) C compiler for the MCU.

```
fsm sendout {
    word packet_data [2];
    state WAIT_DATA:
        wait_sensor (SENSOR_IMU, GET_ACCELERATION);
        delay (1024, GET_HEARTBEAT);
        release;
    state GET_ACCELERATION:
        sint vector [3];
        read_sensor (GET_ACCELERATION, SENSOR_IMU,
        vector);
        packet_data [0] = 1;
        packet_data [1] = (word) calculate_total
        _acceleration(vector);
        sameas SEND_PACKET;
    state GET_HEARTBEAT:
        packet_data [0] = 2;
        packet_data [1] = (word) seconds ();
    state SEND_PACKET:
        address pkt = tcv_wnp (SEND_PACKET, rfd,
        TOTAL_PACKET_LENGTH);
```

```
                  memcpy (pkt + 2, packet_data, 4);
                  tcv_endp (pkt);
                  proceed WAIT_DATA;
        }
```

Figure 1. A sample task in PicOS

For illustration, see the FSM code listed in Figure 1. An FSM definition is a function-like construct beginning with the keyword `fsm` followed by a name. The code is organized into a number of states representing the points where the FSM can be entered (activated). Those states receive symbolic names treated as FSM-relative enumeration constants.

When an FSM is created, its execution commences in the first (top) state. Once an FSM has been activated, it will remain active (holding on to the CPU) until it voluntarily decides to relinquish control. That happens when the FSM executes `release()` or hits the end of the instruction sequence (falling through its closing brace), as for a regular function. Typically, an FSM will do that when it has nothing more to do and it has to wait for an event before proceeding.

Consider the call to `wait_sensor()` in the first state of the FSM. This is a standard PicOS function to await the nearest moment when the indicated sensor triggers an event. It looks like the FSM wants to retrieve acceleration data from the IMU sensor and that the sensor strobes that data by some events (e.g., triggered on an acceleration threshold). While waiting for the sensor event, the FSM also waits for another event (indicated by the call to `delay()`) to be delivered by a timer. The FSM wants to be awakened after the prescribed delay if the sensor event does not materialize in the meantime. The first argument of `delay()` specifies the number of so-called PicOS milliseconds after which the timer event should be triggered. A PicOS millisecond is equal to 1/1024s; thus the timer setting amounts to exactly 1s.

Typically, before releasing the CPU, an FSM executes at least one operation declaring its willingness to respond to an event. A failure to do so is equivalent to termination: the only way for an FSM to sustain itself is to be awakened by events. Formally, the waiting commences at the moment when the FSM releases the CPU. Until then, the FSM can issue more operations identifying possibly multiple (alternative) events that it wants to be awakened by in the future. The effect of those operations is cumulative: the FSM will be awakened by the earliest occurrence of any of the awaited events. When that happens, all the remaining events that the FSM has been waiting for are forgotten.

Any operation specifying an event that the FSM wants to await must provide a state identifier. When the FSM is awakened by the specified event, its code will be entered at the point of the indicated state. Looking at Figure 1, we see that the sensor event will get the FSM to state `GET_ACCELERATION`. If the timer goes off before the sensor becomes ready, the FSM will wake up in state `GET_HEARTBEAT`.

Some operations may block internally. A situation like this corresponds to a system call in a traditional operating system that must block the process until the requisite resource becomes available. Any operation that may ever block also needs a state identifier. When it actually blocks, it will force `release()` internally setting the FSM to be re-activated in the indicated state when it makes sense to try the operation again. This is illustrated with `read_sensor()` in Figure 1. The sensor may need some time to produce the data, during which the CPU can be directed to another FSM. Depending on the circumstances, the operation may immediately succeed, like a simple function call, or fail, in which case the FSM will behave as if it has executed `release()` preceded by an implicit wait request for the event produced by the sensor when the data is ready. This may be different from the event awaited by `wait_sensor()` which indicates an application-level condition. The action of reading the sensor value may involve internal events, similar to an I/O transaction in a traditional system.

There exist operations providing for simple unconditional state transitions. For example, `sameas()` works like a direct goto: it says that the remainder of the code to be executed is the same as the code at the indicated state. The other operation with a similar effect, `proceed()` (see state `SEND_PACKET` in Figure 1), acts as `release()` accompanied by an immediately-triggered event activating the FSM in the specified state. The difference is that the transition via `proceed()` involves a pass through the scheduler loop (Section 3.1.2). The FSM relinquishes the CPU before appearing to the system as ready to enter the new state.

FSMs are expected to be cognizant of the assumption that their operations are reactive. Note that the FSM in Figure 1 preprocesses the 3-vector returned by the IMU sensor before transmitting it in a packet (function `calculate_total_acceleration()`). We do not list this function, but we can imagine that it calculates the magnitude of the vector. It illustrates the kind of acceptable extent of number crunching allowed for an FSM while still maintaining its reactive nature. Normally this nature is stimulated by the very idea of organizing the activities of the program run by the node into a collection of FSMs. Viewed as processes, FSMs are extremely lightweight (the description of an FSM in the system takes just a few bytes, see Figure 4), so a complete program may cheaply consist of many FSMs. Note that an FSM can be created dynamically, for an intermittent task and disappear having accomplished its goal (Figure 2).

An FSM can only be deprived of the CPU between states. While the FSM is allowed to use the stack for its temporary variables, any data stored on the stack is not going to survive state transitions, so such variables are only valid within the temporal context of a single activation of the FSM. This implies two types of local variables: permanent ones, like `packet_data` declared at the top of the FSM code in Figure 1 and temporary ones, like `vector` declared at the beginning of state `GET_ACCELERATION`. The former variable is the facto static: its storage is reserved as for a global variable (or for a `static` variable of a C function). The latter one is automatic (in C parlance) and allocated on the stack (as a regular local variable of a function). Its scope only covers the state in which it has been declared. As the stack of an FSM need not survive state transitions, all FSMs and interrupt service routines can share the same single stack space which is thus never fragmented. Moreover, as the interrupt service routines are not meant to emulate tasks, they can be (mostly) non-interruptible, so the stack has no tendency to run away.

```
fsm blink (sint led) {
    state LED_ON:
        leds (led, 1);
        delay (2048, LED_OFF);
        release;
    state LED_OFF:
        leds (led, 0);
        finish;
}
```

Figure 2. A parameterized FSM

While a given FSM can exist in a number of (possibly dynamic) copies, any permanent local variable declared by the FSM always occurs in a single copy, being effectively shared by all FSMs running the same code. One way to associate a strictly local attribute with a specific instance of an FSM is to use an argument, as illustrated in Figure 2. The FSM is invoked to turn on a LED, specified by the argument and then turn it off two seconds later, before terminating itself.

The private argument of an FSM is the way to tell apart different copies of the same FSM, i.e., to differentiate their actions. It can only be a single simple value (not a structure) amounting to a number or a pointer. The idea is that the default assumption about the extent of the instance-specific information is minimalistic. If there is a need to associate more information with an individual FSM, a structure can be allocated and a pointer to that structure can be used as the FSM argument, so the memory expense is made explicit.

```
...
sint rfd;
...
fsm root {
    state ROOT_START:
        phys_cc1350 (0, MAXIMUM_PACKET_LENGTH);
        tcv_plug (0, &plug_null);
        rfd = tcv_open (WNONE, 0, 0);
        runfsm blink (0);
        runfsm sendout;
}
```

Figure 3. A sample `root` FSM

Every piece of code in a PicOS program is executed as part of an FSM. When the program is started (i.e., when the MCU resets), the system will automatically instantiate a single copy of the FSM named `root` which must be provided by every program. The role of this FSM is similar to the role of function `main()` in a regular C program. Figure 3 shows a simple `root` FSM running the two FSMs from Figures 1 and 2. It starts the two FSMs (operation `runfsm()`) and then quits by failing to issue a wait request in its only state.

3.1.2. Real time aspects

The limited character of multithreading in PicOS raises questions about the real-time responsiveness of the program. The adjective real-time tends to pop up automatically when one mentions embedded systems [57]. Also, the adjective reactive applied to such systems suggests that one should worry about minimizing the response time of the program at least to selected (real-time-conditioned) types of events.

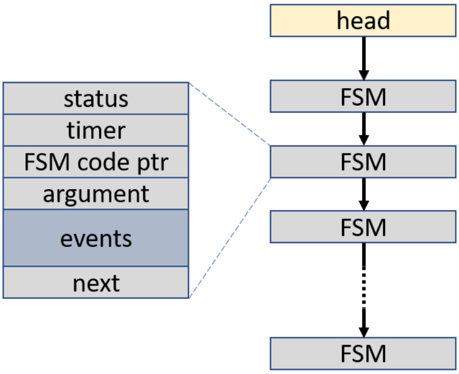

Figure 4. The scheduler queue

In an apparent aggravation of the problem, PicOS scheduler is naively simple. The set of currently existing FSMs is represented by a straightforward list of records depicted in Figure 4. The ordering of the FSMs on the list does not change, except for additions and deletions and reflects the reverse creation time: newly created FSMs are inserted from the front.

The status field of an FSM description indicates whether the FSM is ready to execute and, if this is the case, shows the state where it should be activated. The system executes the FSMs inside the scheduler loop where it scans the list starting from the head looking for the first FSM ready to run. If none is found (which is the most frequent case in a healthy reactive system), the scheduler loop falls through to a HALT instruction to put the CPU into a low-power state awaiting an interrupt.[9] The HALT instruction in fact executes within an inner (tight) loop whose exit condition is set by an interrupt service function when it delivers an event awaited by at least one FSM. When that happens, the (outer) scheduler loop is run from the top.

Note that an event awaited by an FSM can be delivered by the action of another (currently running) FSM or by an interrupt service routine. While interrupt service routines are allowed to run on top of the currently running FSM, the running FSM cannot be preempted by another FSM until it decides to release the CPU. The CPU scheduler is activated in two circumstances:

[9] On the ARM CPU (CC1350), the specific machine instruction is called WFI (Wait For Interrupt).

1. when the currently running FSM releases the CPU,
2. when an interrupt service routine returns, the interrupt has delivered an event rendering some FSMs ready and no FSM was running when the interrupt occurred (the scheduler's control is within the inner HALT loop)

The non-preemptive nature of PicOS FSM threads forces an approach to implementing good responsiveness of the program whereby the critical actions are carried out from interrupt service functions, while the granularity of FSM states still provides for a good response time to the actions buffered by those functions. For example, when receiving serial data from a fast interface, the driver will quickly store the bytes in a buffer on every interrupt (which can freely preempt any FSM) notifying the FSM component of the driver when a complete message materializes in the buffer. Note that with this approach the goals of the interrupt service functions tend to be well defined and the functions themselves tend to be short and simple. Our studies of the real-time responsiveness of PicOS programs have yielded surprisingly good results [58]. As it turns out, many of the real-time requirements posed by applications are naturally fulfilled by PicOS programs with no special attention.

From the viewpoint of the programmer, the advantages of the FSM model are twofold. First, the model simplifies programming and makes the resulting programs error-resistant. The one-state-at-a-time view on the complexity of the actions to be carried out by the program facilitates good comprehension of those actions by the programmer, easy isolation of information to be carried between states and easy identification of potential problems. It also simplifies (to the point of practically eliminating them) all synchronization issues within the application while still providing a fair sense of parallelism and (extremely frugal) multithreading. The other advantage is in enabling natural emulation of distributed systems consisting of large sets of communicating nodes (Section 3.2).

3.1.3. VNETI: the I/O interface

PicOS implements a unified, lightweight and flexible I/O interface, primarily aimed at RF networking, but also useful for other types of communication, e.g., with the OSS over serial devices. Some of its elements can be seen in Figures 1 and 3 which illustrate a complete sequence of incantations needed to expedite a packet over the RF channel.

The interface, dubbed VNETI (for Versatile Network Interface) is shown schematically in Figure 5. To avoid the layering problems haunting small footprint solutions, the interface is essentially layer-less, its semi-complete generic functionality being adjustable by plugins. Physical devices are represented by PHY modules corresponding to device drivers in a traditional system. Multiple plugins and PHY modules can coexist within the same system configuration.

VNETI implements an open-ended management system for pools of packets. A pool can be associated with a descriptor established by the application program and representing a logical connection to some device or channel. The interaction of plugins and PHY modules implements protocols whereby packets can be claimed

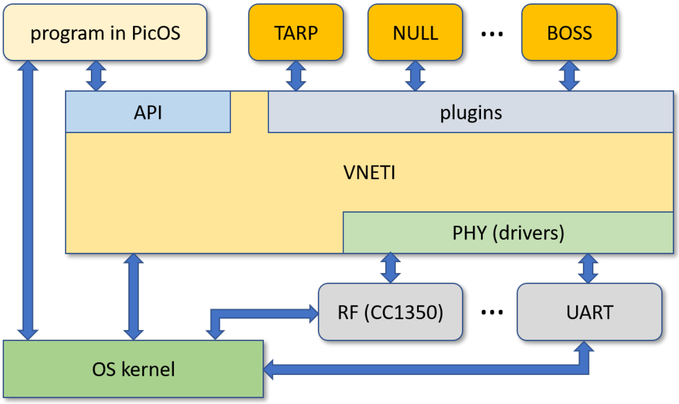

Figure 5. VNETI and its place in the system

by plugins, modified by them, directed towards specific devices, re-processed on timers and so on.

For illustration, the first three statements in the `root` FSM in Figure 3 establish a raw interface to a wireless channel. The first of them sets up a PHY interface to the CC1350 onboard radio assigning it the Id of 0 (the first argument of `phys_cc1350()`). The next operation declares a plugin for VNETI (this is the so-called NULL plugin) and also assigns it an Id of 0 (the Id spaces for the two types of entities are separate). The third operation opens a connection that binds plugin 0 with PHY 0 and returns an integer value interpreted as the descriptor of that connection. The idea is that any packet inserted into the connection by the program will be claimed by plugin 0 which (in this case) will simply pass it to the PHY (move the packet to the transmission pool of the driver). Similarly, any packet received by the PHY will be passed by the plugin to the reception queue associated with the descriptor.

The `sendout` FSM in Figure 1 demonstrates how packets are transmitted; we see this in state `SEND_PACKET`. The action consists of two operations performed by the VNETI functions `tcv_wnp()` and `tcv_endp()`. The first function allocates a buffer for the outgoing packet associating it with the connection descriptor. The `wnp` suffix stands for write next packet and means that the intention (disposition) of the buffer is to be written out. Note that the operation will block if no free memory to accommodate the packet is available at the time. Should that happen, the function will force `release()` marking the FSM as awaiting a memory event with a transition to state `SEND_PACKET`. That event will be triggered when some memory is freed. If the buffer has been allocated successfully, it is filled with the data, with some initial portion of the packet reserved for the PHY header. Finally, `tcv_end()` is called to tell VNETI that the packet is done i.e., its contents have been finalized and the buffer is ready for processing according to its disposition. For a packet allocated by `tcv_wnp()` that means that it should be queued for writing (transmission) by the respective PHY. Once the packet has been transmitted, the PHY will notify the plugin about this event. The plugin's next action will be to instruct VNETI that the packet's buffer can be freed.

Reception (not shown in the FSM) is carried out according to a similar paradigm. Packets received by the PHY over the RF channel are first passed to the plugin which decides whether they should be claimed. The simple action of the NULL plugin is to claim all received packets and instruct VNETI to store them in the reception pool (queue) associated with the connection descriptor. By executing `tcv_rcv()` an FSM can extract the next packet from the reception queue or wait for an event to be triggered when the queue becomes nonempty.

3.2. VUEE: virtual development and testing

VUEE (Virtual Underlay Execution Engine) is an emulator for networks built of nodes running PicOS programs. It has been made possible by the close relationship of the PicOS programming paradigm to SMURPH/SIDE, which is a powerful specification system and simulator for low-level communication protocols [59].

3.2.1. The motivation (a historical note)

PicOS, in its basic concept, was designed as early as 2002 to facilitate a specific project whose objective was to develop a low-cost smart badge equipped with a low-bandwidth RF transceiver allowing it to communicate with similar devices in the neighborhood. Most of our creative effort within the project was spent on the design of the communication protocol for the badges. The protocol was expressed and modeled in SMURPH at the level of detail corresponding to a full realistic implementation. That was possible because a SMURPH specification amounts to an implementation of the protocol in abstract hardware where the protocol program is executed by an event-driven simulator [60]. Specifications in SMURPH look like collections of reactive threads (similar to FSMs in PicOS) which makes their activities naturally expressible as sequences of discrete-time events scheduled by the simulator. The simulator is additionally equipped with tools for modeling the behavior of wireless channels. Channel models can be parameterized to provide for a high-fidelity representation of real-life RF propagation environments [59, 61].

Following the successful virtual validation of the badge protocol, the source code of the model, along with its plain-language description, was sent to the manufacturer to aid in the implementation. Some time later, the manufacturer sent us back their prototype program for the MCU of the badge for our assessment of its conformance to the design. The program had been written for the bare MCU as a single and messy chunk of code trying to approximate the behavior of our high-level multi-threaded virtual implementation via an unintelligible combination of flags, hardware timers and counters. In our struggle to comprehend the code, we concluded that it would be easier to program a simple operating system for the MCU providing a scheduling mechanism mimicking the event-driven kernel of SMURPH and adapt the threads of the model to that kernel retaining their original structure and formal semantics. That was partly inspired by our previous attempts to extend the functionality and application of SMURPH onto controlling real physical systems, as opposed to merely simulating them [62, 63].

For some time, the development of serious applications in PicOS was carried out along two semi-separate paths. Together with the collection of PicOS programs constituting the network's firmware we would build their models in SMURPH to have a virtual vehicle for analyzing, verifying and improving the application without having to run tests with real networks. Owing to the intimate relationship between SMURPH and PicOS the task was not difficult, but it was still tedious because the work was mostly done by hand. Thus, at some point we closed the loop by implementing a compiler, dubbed PiComp, that would automatically produce a SMURPH model (an executable event-driven simulator) from the combined source code of all the PicOS programs destined for the physical nodes. That step forced us to clean up a few messy bits in the original PicOS design and upgrade the methodology of application development [52]. From that point on, applications could be implemented completely virtually, with the final step of flushing the physical devices with the target firmware postponed until we were virtually[10] sure that the (complete) application was ready.

3.2.2. Emulation versus execution

Figure 6 shows schematically the two possible ways of compiling node programs in PicOS. The right path corresponds to the natural case of turning the programs into a collection of flashable image files to be uploaded into physical nodes. The left path outlines the transformation of those programs into a VUEE model.

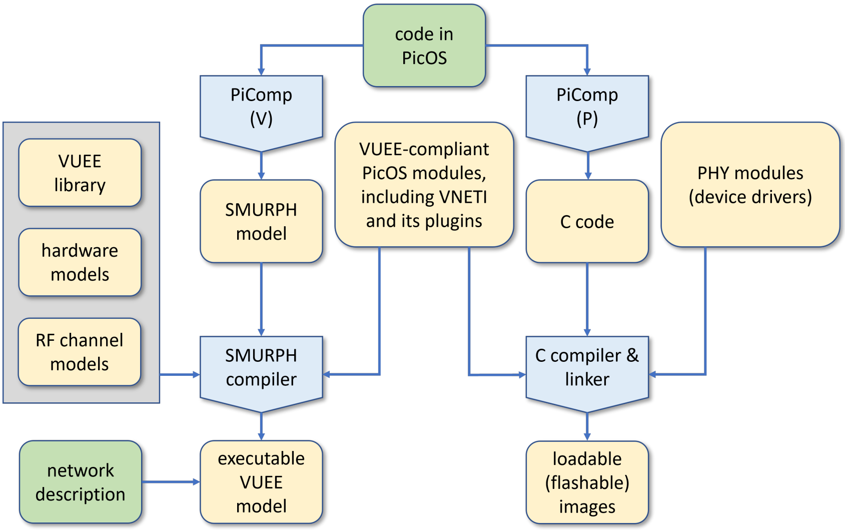

Figure 6. Compilation paths for PicOS programs

[10] Pun intended.

The first stage of either path consists in preprocessing by PiComp. For the right-hand-side path, the compiler converts PicOS-specific programming constructs to C, such that its output looks like a collection of programs in plain C that can be subsequently handled by the MCU C compiler and toolchain. For the left-hand-side path, PiComp turns the collection of programs comprising the firmware for the nodes into a set of source files recognizable by SMURPH as a protocol specification. That involves encapsulating the node programs into C++ methods that can be executed within the context of a virtual object representing a node, such that multiple virtual nodes can coexist and interact within the framework of the composite model. When subsequently processed by the SMURPH compiler the output of PiComp is turned into a single executable program modeling the complete system.

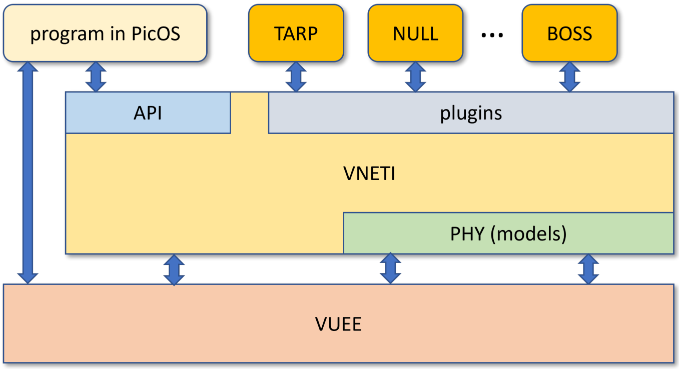

Figure 7. A single-node view of the emulation interface

Many fragments of the PicOS environment (the VUEE-compliant part in Figure 6) occur in a single source version and can be directly compiled by PiComp into either path. The most important of them is VNETI with its set of plugins. As seen by the program of a single node (Figure 7), the emulation layer provided by VUEE interfaces with VNETI on the OS end. The program talks to exactly the same VNETI in its emulated guise as in the real world.

Figure 8 shows the global view. The model encompasses a multitude of nodes, wireless channels and whatever other models of physical components are needed to create a realistic replica of the target environment. Quantitative parameters of the model are described in an input data file submitted to the simulator executable. It is possible to run the same model program for networks of different sizes, with different distributions of nodes, different characteristics of the wireless channel and so on. VUEE models are executed by SMURPH in a fashion that mimics the flow of time in the real network. Serial devices (UARTs) of selected nodes can be expressed as objects mapped to real interfaces. It is thus possible for the WSN model to interact with real-life programs, like OSS components, which can be developed and tested before the physical incarnation of the network can materialize.

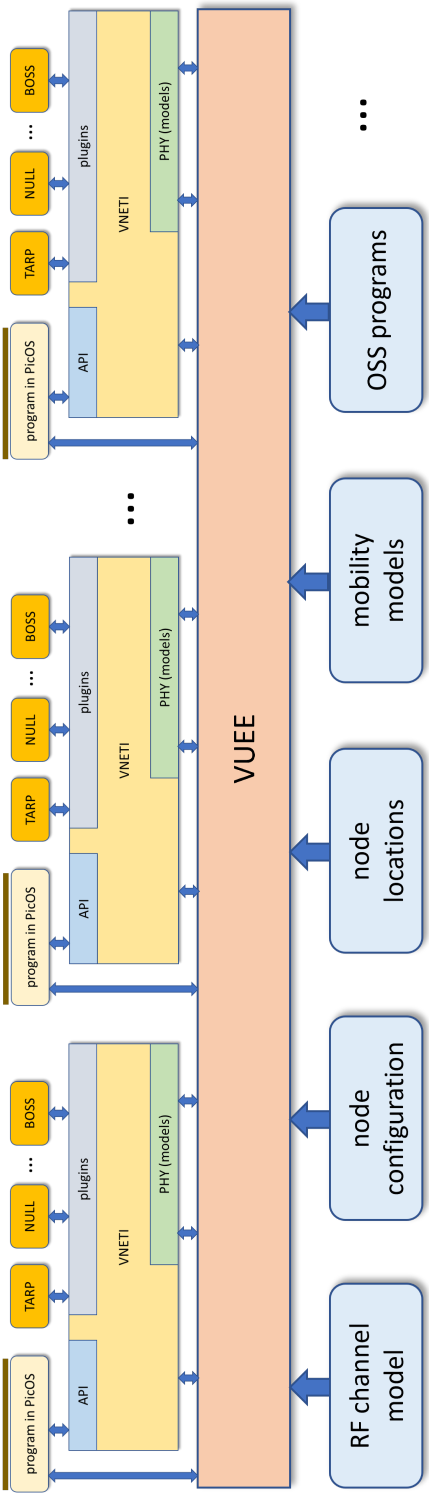

Figure 8. A global view of the emulation interface

3.2.3. Blueprints, praxes and applications

Our way of viewing WSN applications during their development is to treat them as much as possible as purely abstract systems whose design and verification is carried out (almost exclusively) in a virtual environment. Our intention is to come up with a separation of the program semantics into two realms: physical and virtual, along the line of the physical implementation of the sensors and actuators employed by the application. It makes sense to see three levels of this abstraction: blueprint, praxis and application.

By a blueprint we understand the set of node programs isolated from the technical functionality of the sensors and actuators and assuming no particular purpose of the OSS interface. The blueprint captures the open-ended (pure) functionality of the WSN [3]. For example, we may have a data collection network whose end-nodes (to be equipped with sensors) can work with any sensor types that follow some interface standards and whose OSS can do with the collected data whatever it pleases. It is conceivable to implement a complete operation of such a network (in the virtual world) without assuming anything more specific about the interfaces.

A blueprint adapted for a particular function is called a praxis [52]. While the praxis may still abstract from many technical aspects of sensors (like their actual commercial variants), it may have to assume specific sensor types (e.g., IMU, light sensor, push button) according to their intended functionality. It may also have to assume a specific function of the OSS. The adaptation of a blueprint into a praxis may involve some adjustments in the node programs, but those adjustments typically involve static compilation parameters, requiring no replacement for significant fragments of code.

The same blueprint can be used as the basis for many praxes where functionally specific sensors/actuators are going to be mapped to their abstract representatives in the blueprint and customized OSS programs are going to be created to converse with the network about specific requests and data. When the praxis gets to the stage where the PicOS programs can be uploaded into the physical devices, then it becomes an application. We often retain the term praxis in reference to the application. From the viewpoint of the programmer they are the same thing. Blueprints and praxes can be authoritatively developed and tested in a purely virtual environment making their reliable transformations into applications easy and essentially mechanical.

3.2.4. Modeling wireless channels

RF communication in a VUEE model is emulated rather than simulated, in the sense that true packets are transmitted and delivered in the model. The channel model has to account for signal attenuation, interference, bit error rate and so on in a maximally realistic way. In contrast to the standard case of modeling for abstract performance evaluation, we are primarily interested in seeing as much realism in the real-time execution of the model as possible. Our praxes often use the Received Signal Strength Indication (RSSI) as a special sensor for various proximity based events and predicates. One example of such a special sensor is the location tracker, i.e., a subsystem implementing RSSI-based location estimation of the mobile nodes. Having a friendly vehicle for virtual execution of the complete application, one would naturally like to use it for testing/debugging the location tracking component of the OSS [5]. This calls for a high dose of realism in the virtual development setting.

The modeling of physical phenomena related to RF propagation has been very detailed in SMURPH since its inception [64]. In contrast to many popular simulators, where the success/failure of a packet reception is determined once per reception [65], SMURPH provides a detailed model of reception where the opportunities (including interference) may change while the packet is being received. The generic model built into SMURPH is open ended and can be augmented with functions that take into account samples of packet reception events collected from the deployment environment of the real-life network.

4. The network

In this section, we expose the anatomy of a certain blueprint intended to provide a generic WSN basis for advanced data collection applications. The blueprint has been successfully turned into a praxis and, subsequently, into an application in a wireless monitoring system for Independent Living (IL) facilities [5] whose role is to monitor the behavior patterns of patients and track their locations within the campus of the facility.

The blueprint will be referred to as Tags & Pegs (T&P) [3]. Strictly speaking, the name refers to a family of blueprints based on the same generic partitioning of the set of nodes into two main classes, where the functionalities of the actual praxes can differ possibly quite drastically. However, there is no sharp line separating the different blueprints falling under the same paradigm of network organization, because we can always assume that we are looking at a single blueprint suitably parameterized. In our discussion, we shall focus on a well-defined (albeit still generic) variant of the T&P blueprint covering a cohesive subset of the wider concept and on one praxis of the blueprint which will be referred to as Alphanet. This is the patient monitoring network that has been deployed in a number of IL facilities in our collaboration with Alphatronics [5].[11]

4.1. Node types and roles

In many applications of WSNs, one would like to see a semi-infrastructure, built of a subset of the network nodes, even though the network essentially operates in an ad-hoc manner. In a WSN deployed within a facility (a building or a campus) for long-term, sustained operation, such a semi-infrastructure can be affordable, natural and useful. That means that a subset of nodes in the network will be fixed, i.e., preinstalled in selected locations and nailed to the wall. We shall call them Pegs. The other node type will constitute the spontaneous component

[11] http://www.alphatronics.be

of the WSN, e.g., being brought up on demand, exhibiting mobility, collecting data, triggering events. We shall refer to them as Tags.

4.1.1. Constraints

One difference between a Peg and a Tag is that the former can be usually powered from an external power outlet, which means that it is not heavily constrained by its energy budget, while a Tag is typically powered from a battery. In Alphanet, the nodes actually doing the monitoring are worn by the patients (e.g., as wristbands, pendants or badges), or attached to (movable) equipment. The Pegs, on the other hand, are installed in fixed (preferably inconspicuous) places (e.g., electrical enclosure cabinets) and connecting them to solid power sources is usually not a problem.

As we noticed in Section 2.4, the primary energy hog of a node is the RF module. In the context of a mesh WSN this relates to routing/forwarding, i.e., to the problem of maintaining point-to-point connectivity across the network. To be able to forward truly spontaneous traffic, a node must constantly listen to the channel. For a battery-powered node, this will reduce its longevity to days. While frugal implementations of forwarding, based on duty cycling, are possible, they unavoidably reduce the responsiveness of the network. Thus, if the problem can be eliminated by providing a non-exhaustible energy source for the critical subset of nodes, it is natural to take advantage of it.

For a network deployed within a facility (like Alphanet) it is also natural to assume that only the Pegs deal with routing and forwarding while the Tags merely send (and occasionally receive) those messages that originate at (are addressed to) them. Energy-frugal reception is in fact nontrivial because the node can only listen for a message addressed to it within very short, precisely prescribed periods of time (Section 4.1.2).

It is possible to have a network based on a single type of nodes where forwarding roles can be assumed by any node. For example, a (supervised) group of people traveling together, or a set of related goods transported together, may form an ad-hoc network to keep track of the group's well-being. In our view, a network built according to this model is comprised exclusively of Tags assuming both roles, i.e., end devices as well as routers. Such a model is called Routing Tags (RT) [3]. Depending on the activity patterns in the network (and the incurred traffic patterns) the longevity of a battery-powered node can be from days to months, the latter achievable with smart duty cycling. However, certain types of functionality, like fast responsiveness to unscheduled events (alerts), location tracking capability, may not be available or become restricted with their scope trimmed to the energy budget.

All nodes in a given WSN are based on the same (or compatible) MCU and RF module; consequently, their roles can be programmed as flexible. The fact that a Tag (normally) does not act as a router/forwarder need not be hardwired into the node, but it can be determined by a dynamically reconfigurable parameter. This means that, if needed, a subset of Tags from a T&P network can be temporarily reconfigured as an RT network, e.g., to provide for a restricted monitoring service during a field trip.

4.1.2. Communication scenarios

The T&P blueprint implements a number of standard communication scenarios, i.e., types of messages exchanged among nodes. Some of those messages may have to be forwarded. In Section 4.2 we explain how the network implements multi-hop data exchange. We should keep in mind that the exchange is inherently unreliable. This simple fact must be factored into any WSN design meant to be of a practical value. Also recall that the individual messages are short and amount to no more than ca. 40 useful bytes each (Section 2.2).

In a T&P setup, one node in the network acts as the master node providing the network's interface to the OSS.[12] For a data collection network, most traffic originating in the Tags is intended for the master node which will pass the data to the OSS. The communication can go both ways, i.e., a Tag should be able to receive a message from the network, as explained further below.

Messages exchanged in a T&P network comprise the following types:

Master beacon. This message is issued by the master node to notify the network about its identity. Depending on the parameterization of the network, any node may be able to claim the master status. Following the reception of a master beacon, the receiving node learns the identity of its master.

Master beacons are one of the few broadcast messages addressed to all nodes in the network, that is all nodes that can receive the message. A Peg (generally any forwarder node) receiving a master beacon will forward it as a broadcast message, according to the rules outlined in Section 4.2.

Report. A message of this type originates at any node (a Peg or a Tag) and is addressed to the master. The message will be forwarded (Section 4.2) if the master cannot be reached in a single hop. The intention of a report is to notify the master about something that the originating node has discovered. This may be a sensor reading or an event signal from a Tag, or an aggregation packet or a special notification from a Peg, e.g., a virtual sensor event used in location-tracking [5].

The sender of a report may optionally ask for an acknowledgment, i.e., a report may be acknowledgeable or not. In the former case, having received the report, the master will issue an acknowledgment addressed to the original sender. There is no guarantee that the report or the acknowledgment will reach the respective recipients. There is no built-in persistent scheme that would try to implement a reliable channel using the acknowledgment mode for reports. Both the report message and its acknowledgment, as seen by their senders, are (in principle) single-shot transmissions.

[12] Multiple master nodes are possible, but we shall assume in this presentation (for simplicity and clarity) that there is just a single master.

Remote Procedure Call. This is the general case of a message (RPC) sent by one node and addressed to another node, possibly several hops away, neither of the nodes being the master. Nodes may use such messages to implement local communication scenarios that need not involve the master node. An RPC message can be acknowledgeable or not. The idea is the same as for a report, i.e., an RPC message can be viewed as a report addressed to a node that is not the master.

Ping. A message of this kind is not expected to be forwarded beyond its single hop. It is addressed to all nodes (any node) in the neighborhood, as explained below. A ping can be acknowledgeable or not.

It is possible for any node to issue a network-wide broadcast message, but such messages are usually restricted to the master (beacons). A broadcast message puts a strain on the network because it propagates (or at least is intended to propagate) to every node. While there seems to be little use for broadcast messages sent by nodes other than the master, the master beacons play the important role of training the network to efficiently forward reports to the master (Section 4.2.2).

Many (most) of the Tags operate in an energy-frugal regime with their RF modules turned off and only turned on for the brief moment when the Tag sends a report or a ping. Also, the way Tags receive messages (in particular acknowledgments) is different from how it is done by Pegs (which normally listen to the RF channel all the time). When a frugal Tag has something to say, it usually issues a ping. In the simplest case, when the ping does not have to be acknowledged, the RF module is only turned on for the minimum amount of time needed to expedite the ping packet. The idea is that the ping will be handled by any Peg that happens to pick it up. If the ping amounts to a report (e.g., the Tag is reporting a sensor reading or an event), the Peg will transform it into a report message addressed to the master. The Peg will thus act as a proxy of the Tag for conveying the message to the master. Note that the Peg may decide to request an acknowledgment for the report on its own account and dispatch the report persistently until it safely concludes that the master has received the message. If the Tag requires an acknowledgment for its ping, then, immediately following the ping's transmission, it will leave the receiver open for a short amount of time expecting an immediate confirmation packet from the Peg that has picked the ping up. Note that the ping can be received by several Pegs and they all may send their acknowledgments. The Tag only cares about one (any) confirmation which will let it conclude that the message has been passed into good hands. The Tag may retransmit the ping several times if it does not immediately receive a response from a Peg.

The same ping-ACK mechanism is used for passing messages from the network (mostly from the master) to frugal Tags. When the master wants to notify such a Tag about something (e.g., about a new setting of some of its parameters, including a new actuator value), it will send a message to the Pegs that have been recently acting as proxies for the Tag's reports. When any of those Pegs subsequently receives a ping from the Tag, it will insert into the ACK window the message from the master. The message may force the Tag into a temporary high-duty listen mode, if the Tag is likely to receive more messages in the nearest future. The Tag may put itself into this mode automatically, e.g., when it detects and signals a special condition, in expectation of a related response. This makes particular sense when the Tag controls actuators.

RPC-style communication may facilitate data aggregation or local collaborative scenarios where the intermediary role of the master is not needed or would be inconvenient (e.g., creating a bottleneck). Message exchange in such scenarios typically proceeds both ways, so the traffic is (automatically) acknowledged.

4.2. TARP: multi-hop communication

Our networks feature a generic and holistic forwarding scheme operating in a layer-less manner. The scheme comes under the name TARP (Tiny Ad-hoc Routing Protocol) and is implemented as a plugin to VNETI (Section 3.1.3). TARP does not belong to any particular networking layer and, in the literal interpretation of its holistic spirit, has no layers inside.

4.2.1. Forwarding by the rules

When transmitted or forwarded, a packet in TARP is essentially broadcast to all neighbors of the transmitting node. There is no indication of its explicit next-hop handler and there is no data-link-layer encapsulation of the packet. The question asked by a node picking up a packet and seeing that it is destined elsewhere is: Do I have a reason not to forward the packet?. The form of this dilemma implies that the default action of the node is to forward (retransmit) any packet that comes its way and appears to be addressed elsewhere. This altruistic behavior must be constrained to prevent infinite flooding and the efforts of a scheme built along these lines will focus on reducing the extent of the default altruism, as opposed to finding grounds for bringing it in.

The unifying feature of practically all popular schemes for mesh communication is the explicit notion of a route, understood as a specific path in the network and the consequent data-link-layer encapsulation of the information transmitted in packets. Even if, in a recognition of the dynamics and fuzziness of paths in the unkempt environment of WSNs, some protocols attempt to provide multiple, alternative routes [66, 67], the path concept (and the data-link-layer encapsulation) is always there. In our opinion, this is the reason why those schemes consistently fail in any practical application over a distance of more than one or two hops. [68] A node operating under TARP is interested in acquiring information that will let it reduce the innate eagerness to forward everything that comes its way. Note that a node deprived of that information, e.g., because it has run out of memory, has just joined the network, or has moved to a new neighborhood, will be contributing to the collective task of passing the packets through. Its operation may appear redundant (and wasteful) for a while, but its uninformed status will have no negative impact on the overall connectivity. Contrast this with the behavior of a node operating in the strict fixed-path forwarding regime that has run out of memory to store the routing information or has moved to a new place. Until the problem is detected, diagnosed and remedied via an appropriately orchestrated action, the node will be completely useless as a forwarder.

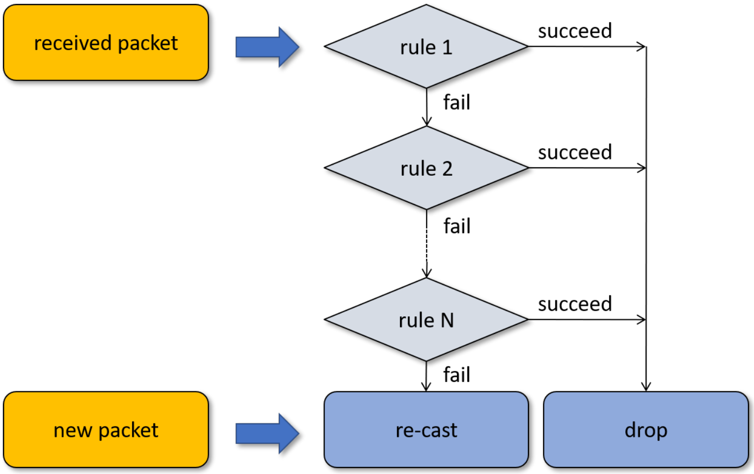

Figure 9. The mechanism of TARP rules

TARP is implemented as a series of rules [69] that are applied by a node in sequence to every received packet, as shown schematically in Figure 9. By receive we understand physical reception (perception) by the node's RF module; it does not mean that the packet is destined for the node.

The rules are executed in sequence, starting from the top and the first rule that succeeds stops the process and causes the packet to be dropped. Note that by a success we mean finding a reason why the packet should not be forwarded.

A rule always has access to the complete packet and it can use whatever criteria it finds useful to make the decision. Among the operations that a rule may want to execute is actual packet reception, i.e., passing the packet to the application. A rule is also allowed to modify the packet's contents before forwarding it, as well as create a new packet and inject it into the neighborhood. Some rules (Section 4.2.2) assume standard contents of the packet (header), but the full set of rules in TARP is open to the application which is free to establish whatever rules best fit its traffic patterns. For example, the application may opt for associative communication [51] where nodes have no explicit addresses.

4.2.2. The standard rules

Notwithstanding the freedom in building the rules by the application, some TARP rules are considered standard by virtue of their being adopted (possibly with differentiating parameters) by our T&P and RT blueprints. They are aimed at facilitating traffic patterns fitting the message types discussed in Section 4.1.2. Those patterns assume that nodes are addressable with numerical identifiers, as

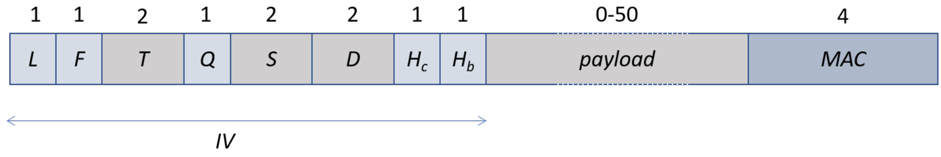

Figure 10. Paket format assumed by the standard set of rules

in most networks. All packets feature the standard format shown in Figure 10. The numbers above the fields indicate their length in bytes.

The fields have the following meaning:

L This is the packet length, i.e., the number of the remaining bytes in the packet. The field is required and automatically interpreted by the RF module.

F This field identifies the packet type. It consists of a 5-bit class identifier plus 3 binary flags.

T The time stamp of the packet inserted by the sender and reflecting its local second-grained clock modulo 2^{16}. This clock is synchronized to the master clock based on the beacons received by the node.

Q This is the packet's serial number (modulo 256) assigned by the source. Consecutive packets sent by the same source node have their serial numbers incremented by 1.

S The node address of the packet's sender (source). Node addresses are between 1 and 65535.

D The node address of the packet's destination, or 0 for a broadcast packet.

H_c This is the so-called forward hop count of the packet, as explained below.

H_b This is the so-called backward hop count inserted by the packet's sender S, as explained below.

MAC This is the message authentication code, a cryptographically secure checksum assessing the packet's integrity.

One of the simplest possible rules is called LHC and its role is to limit the number of hops that a packet may experience. When a packet is created by its source S, the H_c field is set to zero. Then, on every forwarding (including the first transmission by S) H_c is incremented by 1. When the rule sees that H_c has reached the maximum H_{max} (defined by the network), it will succeed and the packet will not be forwarded.[13]

Another rule, dubbed DD (for Duplicate Discard), tries to avoid forwarding the same packet more than once. This is where the serial number Q comes into play. Having retransmitted a packet, the forwarding node caches its signature which amounts to the combination of the sender address S and the serial number

[13] We shall see shortly that H_c (as a forward-increasing hop counter) is also needed by another rule. This is why we prefer it to a decreasing counter decremented towards 0.

Q. The rule fails if the signature of the packet about to be forwarded is already present in the cache.

The two rules mentioned above are the standard countermeasures against uncontrolled flooding used as the simplest way of disseminating information in a WSN [70]. While one may be willing to put up with the redundancy of flooding when the goal is in fact to reach all nodes of the network, it does not appear as an efficient scheme for point-to-point communication. The next rule is a step in this direction. The rule is named SPD (for Suboptimal Path Discard) and, similar to DD, its operation is based on caching some information overheard by the node.

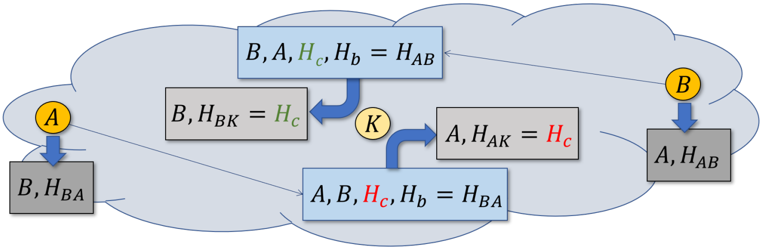

Figure 11. A scenario for the SPD rule

Suppose that some node K overhears a packet sent by some other node A. K learns that the number of hops via which it can be reached from A is H_c. Thanks to the DD rule (operating before SPD), the packet is extremely likely to be the first copy reaching K, as any lagging and wandering retransmissions arriving over less opportune (longer) routes would have been eliminated as duplicates along the way. Thus K may assume the H_c represents the current minimum number of hops separating it from A. K puts this information, represented by the pair $< A, H_{AK} = H_c >$, into a local cache.

This kind of operation is carried out by all nodes in the network as they pick up packets in their neighborhoods. After a node has been around for a while, it will have accumulated the last-best hop information for all the nodes whose packets it has overheard, as much as can be fit into the cache. When the node dispatches a packet addressed to some destination D, it will insert into the H_b field of that packet the last-best hop count from D extracted from its cache. The value of H_{max} is used, as a synonym of unknown if the information is not available.

Suppose that, having intercepted a packet sent by A and addressed to B (as in Figure 11), node K executes SPD for the packet to decide whether it should be retransmitted or dropped. Suppose that an entry for B is available in K's cache. The rule calculates $H_t = H_c + H_{BK}$ and compares H_t with H_b extracted from the packet's header. If $H_t > H_b$, the node may suspect that its assistance will be redundant. This is because the estimated number of hops separating A and B is H_b, but when forwarded by K the packet is expected to make more hops.

The rule is driven by two parameters accounting for the fuzzy nature of the cached information and the possibility of its being outdated. The first of those parameters is slack interpreted as the acceptable excess of the path length over the currently estimated minimum. The rule will succeed if $H_t > H_b$+slack and fail otherwise. The second parameter, dubbed relax, controls a mechanism preventing possible lockouts caused by stale data. Every cache entry includes a counter incremented whenever the rule succeeds (i.e., drops a packet) on account of the entry and reset when the rule fails. The value of that counter is divided by relax and the integer result of that division is added to the right-hand-side of the above equation turning it into $H_t > H_b + \text{slack} + \lfloor \text{counter}/\text{relax} \rfloor$. When relax is set to zero, the mechanism is disabled, i.e., the extra term incorporating relax is assumed to be zero.

At first sight, setting slack=0 may seem like a way to enforce shortest path forwarding, at least after the nodes have been given time to fill their SPD caches with up-to-date information. Note, however, that the paths are not strictly prescribed and the forwarding may go different ways on different occasions, even when the cache contents remain unchanged. This is because of the nondeterminism of simultaneous retransmissions and the operation of DD. Even with slack=0, there may be (and usually are) multiple shortest paths passing through different (alternative) nodes in the neighborhood reachable over the same (locally minimum) number of hops from a given forwarder. This causes redundancy which is sometimes useful, because it improves the Packet Delivery Fraction (PDF) under light load, but can be detrimental if unchecked. This is why the standard set of rules includes one more mechanism aimed at reducing the number of redundant retransmissions over parallel and equally opportune paths.

Imagine two nodes within the mutual transmission range deciding to forward the same packet. The nodes have received a copy of the packet (at the same time), they have run the SPD rule and the rule has failed at both nodes. Each of them concludes that it offers the smallest possible number of hops to the destination and invokes VNETI (Section 3.1.3) to queue the packet for (re)transmission. The driver implements a simple collision avoidance scheme based on LBT (Section 2.2) where it listens to the channel for a short while and possibly backs off randomly, before commencing the transmission of the queued packet. Statistically, one of the two packets will go first, with a good likelihood of preempting the other packet without a collision. The packet transmitted by the quicker node is likely to be subsequently intercepted by the other one which has rescheduled its transmission waiting for the channel to become idle. There is a rule, named SPP (for Simultaneous Path Preemption) which will examine the packet after reception and scan the queue of packets awaiting transmission. If the packet is found in the queue (based on its signature), it will be deleted, thus its simultaneous transmission, now considered redundant, will be prevented.

Note that SPP must be run before DD because the latter rule would have succeeded on the packet that has won the LBT race (thus eliminating it from the view of the further rules in the chain). The role of SPP is not as much to ignore the intercepted packet (although it succeeds when the packet's copy is found in the transmit queue) as to erase its copy queued for retransmission. The kind of access to the packet queue (conveniently provided by VNETI) illustrates the advantages of the layer-free approach to wireless networking [71, 72].

4.3. Reliability and resilience

A comprehensive and convincing performance study of our networks is not possible in a single paper, owing to the multitude of conditions affecting real-life deployments. For example, two sister installations of essentially the same Alphanet [5] exhibited drastically different behaviors because the construction materials used in one of the buildings turned out to be maliciously non-conductive to RF propagation.

Constructing a serious communication scheme, intended for reliable and resilient WSNs, as a derivative of flooding may raise brows in people obsessed with performance guarantees. Note however, that schemes based on solid routes also resort to flooding at some stages, if only to discover the neighborhoods and construct the explicit routes [73]. In our opinion, the only way to accrue any luck at all with multi-hop ad-hoc networking is to start with the assumption that nothing is guaranteed. Then one can plan for reinforcing configurations of unavoidably flimsy events that together conspire to bring about a useful likelihood of success. This is exactly what TARP has been designed to accomplish.

The lack of an explicit data-link layer in TARP precludes collision avoidance mechanisms based on neighbor-to-neighbor handshakes of the RTS-CTS flavor [74] and synchronization techniques assuming that the intention of a packet sender is to hit a single specific recipient [75]. RTS-CTS handshakes have been exposed to criticism. While they may work to some extent in environments with long packets, they are in fact harmful when packets are short [76, 77]. In a dense mesh network, false blocking on exposed terminals may become problematic as well [78].

4.3.1. Fuzzy acknowledgments

One may put forward an argument that with an explicit handover of the packet to a specific next hop neighbor, it is possible to increase the reliability of the hop by resorting to acknowledgments and retransmissions. Strict hop-by-hop acknowledgments are not possible in TARP, to its apparent disadvantage. There is, however, an optional mechanism that a forwarding node can employ to increase the reliability of its assistance. Having retransmitted a packet, the node, call it K, can hold on to it until it has heard the same packet (identified by the signature, Section 4.2.2) retransmitted by one of the nodes in the neighborhood. By looking at H_c in the overheard packet, K can tell if the packet has made progress, which can be taken by K as an indication that its mission has been fulfilled. If the packet's echo does not materialize, K may retransmit the packet a few times, at some intervals. This mechanism, dubbed fuzzy ACKs, has two parameters: the retransmission interval fdelay and the maximum number of additional retransmissions fretr (fretr $=0$ means fuzzy ACKs disabled).

One little twist needed to make the idea work is to force the final destination of the packet to also forward it, so the last forwarder(s) can be assured about the success of their operation. It is enough to retransmit a dummy version of the packet only consisting of the header: the required information amounts to S and Q (the signature) plus H_c.

Somewhat contrary to the intuition of improving the reliability of TARP's broken data-link layer, fuzzy ACKs are not very useful in practice. The only class of scenarios where they prove helpful involve bottlenecks, i.e., configurations where the connectivity of separate sections of the network is channeled through a narrow stripe of a few critical nodes. Then, the fuzzy ACKs basically emulate true data-link-layer acknowledgments along the narrow critical path.

4.3.2. The efficiency of TARP paths

While blanket statements about the performance of mesh WSNs are usually meaningless, the analysis of examples of their behavior may go a long way towards explaining their important features. Unfortunately, such examples are usually difficult to collect because taking snapshots of activities within massive real-life networks is not feasible.

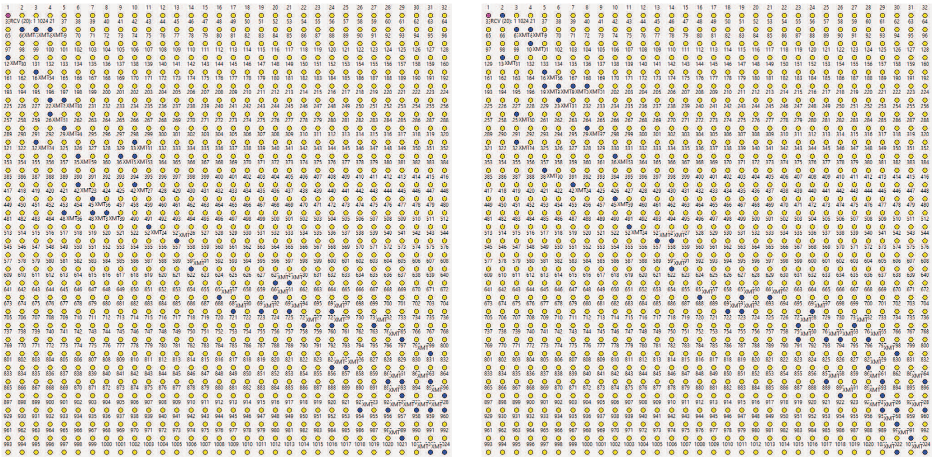

Figure 12. Two sample paths across the network

In our case, thanks to VUEE and the realism of its RF channel models [3, 52], we can look closely into the interiors of quite large networks in authoritative emulations. Figure 12 shows two random paths taken by a packet in a 1024-node regular network, forming a 32×32 square, traveling across the diagonal, to the master node (node number 1) located in the left-upper corner. The propagation model corresponds to CC1350 operating within the 916 MHz band at the raw rate of 38,400 bps. The length of the packet transmitted from the right-bottom node (number 1024) to the master is 31 bytes (16 bytes of payload + 15 bytes of framing, see Figure 10). The network is deployed in an open and plain field with the nodes placed about 1 m above the ground. The effective communication range is about $100m$ (Figure 13). The nodes are regularly spaced on the grid with the separation of 40m along either dimension. TARP is configured to operate with these rules: LHC, SPP, DD, SPD, with slack$=1$ and relax$=0$ (Section 4.2.2). Fuzzy ACKs are disabled.

The network has been on for a while, which means that the master has had an opportunity to issue several beacons that have reached all nodes. Note that when a node receives a beacon packet, it sets an entry in the SPD cache with the value of H_{MK} indicating the current minimum number of hops separating it from the master.

The highlighted nodes are those that have retransmitted the packet sent by node 1024 on its way to the master. Although the two paths have been obtained under the same contents of the SPD caches at all nodes (no new beacon has been issued in the meantime), they are different. The diversity is caused by several random factors, including LBT delays and packet losses. Different nodes forwarding the same packet may preempt one another in many different, inherently random, ways. The SPP rule will cancel some of those transmissions before they materialize and the DD rule will trim out the late packets. A longer hop may sometimes succeed by accident even if the likelihood of success is low and the hop would be considered unreliable under point-to-point forwarding. The DD rule always kicks in in such cases, taking advantage of any flukes, while keeping the more reliable forwarders ready to seamlessly help in less lucky scenarios.

Note that with slack $=1$, SPD admits one-hop detours with respect to the estimated minimum number of hops, which offers a significant number of alternative paths. The vast majority of those paths are trimmed down by DD and SPP. The redundancy is partially actual: the number of nodes involved in passing the packet from the source to the destination is roughly twice as large as the number of hops taken by the packet and partially potential: some transmissions have been preempted and we do not see all the nodes that were standing by and willing to help.

Instead of becoming disappointed with the actual redundancy, let us do a simple calculation comparing it to what we could reasonably expect from a point-to-point scheme trying to address the same forwarding problem. The average number of hops experienced by a packet traveling from node 1024 to the master (as seen in Figure 12) is ca. 21. The absolute minimum is probably around 11 hops, as suggested by the few sparse spots in the paths. Figure 13 shows the observed Packet Delivery Fraction (PDF) across different (feasible) hops within the grid measured under conditions of light background noise. The PDF across the longest of the hops marked in Figure 13 is ca. 65% translating into the loss rate of 35%. The probability of success across 11 consecutive hops like this is less than 0.01 and no point-to-point forwarding scheme would bet on such a path. A rigid route consisting of 21 steps may be built in many ways, e.g., of 10 long hops (112.8m) and 11 short ones (56.4m). Using the numbers from Figure 13, one can calculate the expected number of all transmissions needed to deliver the packet to the other

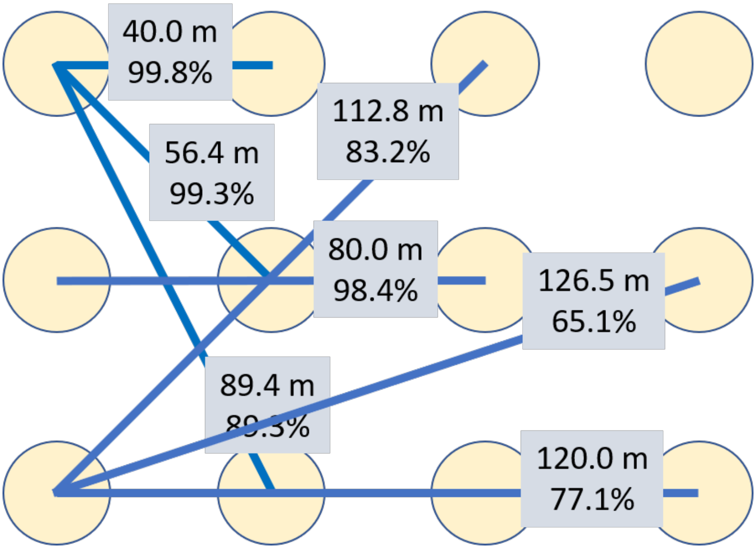

Figure 13. Packet delivery fraction for different hop lengths across the grid

end. Whichever reasonable way the path is split into 21 hops, that number comes out between 45 and 62 which exceeds the number of highlighted nodes seen in Figure 12. Moreover, in a point-to-point route the overhead is inflexible: when the forwarding fails for reasons beyond a casual packet loss (e.g., one of the nodes goes down or disappears), the redundancy of the data-link ACKs will not help until the route is rebuilt. In TARP, the redundancy is automatically (and mostly invisibly) spread over the neighborhood, so it kicks in automatically when the previous best route becomes unavailable, for whatever reason.

4.4. Security

Three security aspects of WSN applications are taken care of by TARP: integrity, confidentiality and resistance to denial of service (DoS) attacks [79]. Regardless of the nature of the application, the integrity problem comes as the first issue to be addressed. Even if the application (and the WSN) carries no confidential traffic, it should be resistant to simple attacks, like pranks, where unsophisticated intruders might be able to insert fake (or duplicate) packets into the network or forge formally legitimate traffic causing confusion and possibly havoc.

4.4.1. Confidentiality and integrity

Security tools adopted in a WSN built of small-footprint devices should be computationally inexpensive and commensurate with the capabilities of the MCUs [32, 80, 81]. In contrast to the general and blanket view on the security issues of WSNs or IoT networks [82], we assume that our networks are holistically designed for their applications [52, 83]; thus, our tools can be simple and configurable on the per-praxis basis. Besides, they should be optional and the option should not overtax the resources.

While security naturally connotes confidentiality, most networks that are expected to be strongly secure do not demand that the traffic carried by the network be confidential. For example, a WSN assisting in disaster management may have to be resilient and secure in terms of the integrity of its traffic, but the secrecy of the messages may not be important, even if the disaster has been caused by a malicious (e.g., terrorist) act. The indications of the network's sensors, as well as the settings of its actuators, may be easily deducible by looking at the scene and plainly available for everybody to see, but the actual readings and settings must be conveyed reliably and securely to the respective points in the network.

Both confidentiality and integrity in TARP are based on AES [84]. The nodes are preset with the key (or possibly multiple keys) when the network is prepared for physical deployment. CC1350 comes outfitted with an integrated fast AES module operating at the bandwidth of 118 Mb [29]; thus, the impact of encryption/decryption on packet processing time is negligible.

If the packet payload has to be encrypted (for confidentiality) a special flag is set in the F field of the packet header (Figure 10). The entire packet header, denoted by IV in Figure 10, is then used as the initialization vector (to diversify the encryption of the same payload) and ciphertext stealing [85] is applied to produce an encrypted version of the payload with the same length as the original. Note that this is only possible if the original length of the payload is at least 16 bytes which is the block length for AES. Therefore, the minimum payload length for packets whose payloads have to be encrypted is 16 bytes. This is one argument against encrypting all payloads unconditionally, despite the negligible cost of AES encryption on CC1350.

In special cases, the payloads of packets addressed to specific destinations (but not broadcast packets) can be encrypted with destination-specific keys. This way the packet payloads may be rendered unreadable at any nodes other than the sender and the destination (both parties must share the key). Note that this may preclude the application of custom rules based on information not available in the packet header.

Regardless of the optional payload encryption, the integrity of every packet is guarded with the MAC code [86] appended as the packet's trailer. The code is generated by concatenating the IV with the packet payload (after its optional encryption) and encrypting the concatenated string of bytes padded with zeros to a multiple of 16 bytes. The code is then obtained as the first 4 bytes of the resulting ciphertext.

The T field of the packet header (Figure 10) is filled by the source node with its second-grained time stamp modulo $2^{16} = 65536$. The second clocks of all nodes are synchronized by the master via the beacon which carries the full 32-bit second clock reading of the master. The synchronization is not meant to be perfect. One role of the T field is to diversify the IV space for MAC calculation and for the optional payload encryption. Its second role is to prevent attacks against network integrity where an attacker could record legitimate packets and play them back later to confuse the network or force it to carry out mischievous actions. A node receiving a packet whose time stamp deviates too much from its own time stamp will just ignore the packet. By the very nature of TARP, a formally correct packet arriving at the node slightly out of time cannot be harmful. If it happens to be obsolete on account of the DD rule, it will be dropped as a duplicate. An attacker playing back fresh packets will be actually helping the network, acting as an extraneous regular node.

4.4.2. DoS resistance

Denial of service attacks against WSNs may come in different flavors [87–89]. The absence of a data-link layer in TARP precludes many of them, like fake RTS/CTS packets, hello floods, ACK spoofing, dummy connection requests and, generally, any attacks trying to fool the (non-existent) medium access control/data-link layers. Assuming the integrity enforcement mechanism described in Section 4.4.1, the only conceivable attack type (to which all wireless networks are exposed) is jamming. Some realizations of the physical layer (spread spectrum, frequency hoping, redundant encoding, filtering) may render jamming more difficult. TARP is largely transparent to such schemes, so they are applicable in our case (e.g., redundant encoding can be selected as an option in the CC1100/CC1350 driver [29, 30]).

A jamming attack is symptomatically equivalent to the disappearance of a node or a group of nodes. In some applications of our network, notably for assistance in disaster management and recovery, the nodes may be considered disposable and possibly large fragments of the network may be destroyed as part of the assumed functionality. The network's resilience to jamming attacks can thus be re-interpreted as a measure of its endurance in the face of predictable damage, where the goal is to maintain connectivity for the maximum amount of time using whatever nodes still remain in the game.

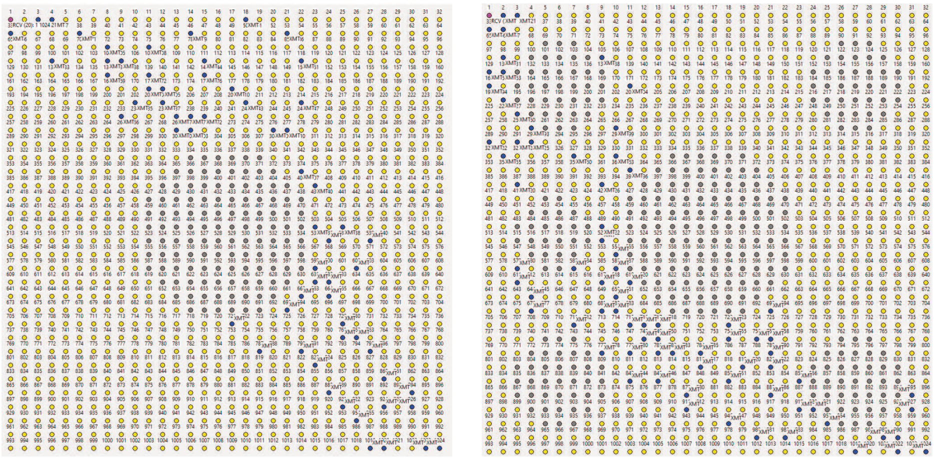

Figure 14. Forwarding around holes

Section 4.3.2 and Figure 12 give us grounds to expect that TARP networks will be responsive to changes in the forwarding opportunities and at least local problems are well taken care of by the distributed redundancy of paths. Figure 14 has been obtained under the same conditions as Figure 12, except that the grayed-out (circular) subsets of the nodes have been disabled. The packets sent by the extreme right-bottom node have made it to the master despite the holes blocking their way. It should be noted that the feat has been accomplished under the same knowledge as for Figure 12, i.e., the SPD caches have not been refreshed by a new beacon from the master. The packet delivery fraction is about 80% for Figure 14a and about 70% for Figure 14b.

Of course, once the master beacon is circulated through the network, the impact of the holes will be greatly reduced, at least for as long as the network remains formally connected. As the beacon floods the network, its frequency is kept at the minimum required for acceptable operation (and is usually expressed in minutes or tens of minutes).

Notably, the good response to damage is obtained under moderate relaxation of the SPD rule, such that the actual redundancy of paths (Figure 12) is acceptable, even by the standards of point-to-point forwarding schemes. If drastic damage scenarios are anticipated, the parameters can be relaxed to smoothly accommodate even more destruction without compromising the connectivity of the remaining subset of the network.

5. Summary

We have presented a holistic WSN concept built of three components integrated into a platform for developing complete wireless sensing applications. The components are:

PicOS An operating system for small-footprint MPUs equipped with tools for constructing compact, multithreaded, energy-aware, reactive, networked programs where collections of wirelessly-connected motes can execute distributed applications (praxes) implementing sophisticated data collection and dissemination systems.

VUEE A system for specification and virtual execution of distributed reactive programs emulating collections of computing devices interconnected over wireless channels. The system provides for detailed and realistic expression of the properties of real-life wireless channels needed for complete virtual specifications of PicOS praxes.

TARP A meta-protocol where multi-hop communication in wireless networks built of resource-frugal devices is described via sequences of rules applied to packets overheard by those devices. TARP facilitates instinctive collaboration of all nodes compensating for the inherent unreliability of the wireless medium, which in the context of massive WSNs is additionally aggravated by the resource-limitation and fragility of the individual nodes.

We have shown how the three components are harnessed together to building networked applications where the WSN nodes can effectively and efficiently communicate over many hops under conditions of uncertainty pervading ad-hoc wireless systems.

The programming paradigm of PicOS, combined with the layer-less communication interface, results in small programs expressed in a self-documenting manner and resilient to programming errors and misinterpretation of the reactive semantics of the target program to its specification. Such programs are easy to execute in a virtual environment behaving as an event-driven simulator where they can exhibit the full range of behaviors of the complete real-life application. This renders the process of their development friendly to the programmer, easy, dependable and convenient.

References

[1] Lu C, Saifullah A, Li B, Sha M, Gonzalez H, Gunatilaka D, Wu C, Nie L and Chen Y 2015 *Real-time wireless sensor-actuator networks for industrial cyber-physical systems*, *Proceedings of the IEEE* **104** (5) 1013

[2] Kabadayi S, Pridgen A and Julien C 2006 *Virtual sensors: Abstracting data from physical sensors*, *International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM'06)*, IEEE

[3] Gburzyński P and Kopciuszewska E 2019 *On rapid development of reactive wireless sensor systems*, *System Safety: Human, Technical Facility, Environment* **1** (1) 574

[4] Silberschatz A, Galvin P B and Gagne G 2014 *Operating system concepts essentials*, Wiley Hoboken

[5] Gburzynski P, Olesinski W and Van Vooren J 2016 *A WSN-based, RSS-driven, real-time location tracking system for independent living facilities*, *DCNET* 64

[6] Conti M and Giordano S 2013 *Multihop ad hoc networking: The evolutionary path*, *Mobile Ad Hoc Networking* **35** (3)

[7] Gubbi J, Buyya R, Marusic S and Palaniswami M 2013 *Internet of things (IoT): A vision, architectural elements and future directions*, *Future generation computer systems* **29** (7) 1645

[8] Hansen C J 2015 *Internetworking with Bluetooth low energy*, *GetMobile: Mobile Computing and Communications* **19** (2) 34

[9] Gill K, Yang S H, Yao F and Lu X 2009 *A ZigBee-based home automation system*, *IEEE Transactions on consumer Electronics* **55** (2) 422

[10] Buratti C, Conti A, Dardari D and Verdone R 2009 *An overview on wireless sensor networks technology and evolution*, *Sensors* **9** (9) 6869

[11] Andreev S, Petrov V, Dohler M and Yanikomeroglu H 2019 *Future of ultra-dense networks beyond 5G: harnessing heterogeneous moving cells*, *IEEE Communications Magazine*

[12] Oracevic A, Akbaş S, Ozdemir S and Kos M 2014 *Secure target detection and tracking in mission critical wireless sensor networks*, *International Conference on Anti-Counterfeiting, Security and Identification (ASID)*, IEEE 1

[13] Krivtsova I, Lebedev I, Sukhoparov M, Bazhayev N, Zikratov I, Ometov A, Andreev S, Masek P, Fujdiak R and Hosek J 2016 *Implementing a broadcast storm attack on a mission-critical wireless sensor network*, *International Conference on Wired/Wireless Internet Communication*, Springer 297

[14] Rani S and Ahmed S H 2015 *Multi-hop routing in wireless sensor networks: an overview, taxonomy and research challenges*, Springer

[15] Ai Y and Cheffena M 2017 *On multi-hop decode-and-forward cooperative relaying for industrial wireless sensor networks*, *Sensors* **17** (4) 695

[16] Chen C-P, Chuang C-L and Jiang J-A 2013 *Ecological monitoring using wireless sensor networks: Overview, challenges and opportunities*, *in Advancement in Sensing Technology*, Springer 1

[17] Lloret J, Garcia M, Bri D and Sendra S 2009 *A wireless sensor network deployment for rural and forest fire detection and verification*, *Sensors* **9** (11) 8722

[18] Chowdary V and Gupta M K 2018 *Automatic forest fire detection and monitoring techniques: A survey*, *Intelligent Communication, Control and Devices*, Springer 1111

[19] Reina D, Askalani M, Toral S, Barrero F, Asimakopoulou E and Bessis N 2015 *A survey on multihop ad hoc networks for disaster response scenarios*, *International Journal of Distributed Sensor Networks* **11** (10) 647037

[20] Lee S-H, Lee S, Song H and Lee H-S 2009 *Wireless sensor network design for tactical military applications: Remote large-scale environments*, *MILCOM 2009-2009 IEEE Military communications conference*, IEEE 1

[21] Kumar K A, Krishna A V and Chatrapati K S 2017 *New secure routing protocol with elliptic curve cryptography for military heterogeneous wireless sensor networks*, *Journal of Information and Optimization Sciences* **38** (2) 341

[22] Jiang H, Chen L, Wu J, Chen S and Leung H 2009 *A reliable and high-bandwidth multihop wireless sensor network for mine tunnel monitoring*, *IEEE Sensors journal* **9** (11) 1511

[23] Alrajeh N A and Lloret J 2013 *Intrusion detection systems based on artificial intelligence techniques in wireless sensor networks*, *International Journal of Distributed Sensor Networks* **9** (10) 351047

[24] Chaudhari B S, Zennaro M and Borkar S 2020 *LPWAN technologies: Emerging application characteristics, requirements and design considerations*, *Future Internet* **12** (3) 46

[25] Wixted A J, Kinnaird P, Larijani H, Tait A, Ahmadinia A and Strachan N 2016 *Evaluation of LoRa and LoRaWAN for wireless sensor networks*, *IEEE SENSORS*, IEEE 1

[26] Mekki K, Bajic E, Chaxel F and Meyer F 2019 *A comparative study of LPWAN technologies for large-scale IoT deployment*, *ICT express* **5** (1) 1

[27] Davies J H 2008 *MSP430 microcontroller basics*, Elsevier

[28] Bhadra D and Stevens K S 2017 *Design of a low power, relative timing based asynchronous MSP430 microprocessor*, *Design, Automation & Test in Europe Conference & Exhibition*, IEEE 794

[29] Texas Instruments 2014 *CC1350 SimpleLink Ultra-Low-Power Dual-Band Wireless MCU*, *Technical document SWRS183B*

[30] Texas Instruments 2014 *CC1100 Single Chip Low Cost Low Power RF Transceiver*, *Technical document SWRS038D*

[31] Olsonet Communications Corporation 2014 *CC1100 Driver Notes*, *PicOS Documentation*

[32] Karlof C, Sastry N and Wagner D 2004 *TinySec: a link layer security architecture for wireless sensor networks*, *Proceedings of the 2nd international conference on Embedded networked sensor systems*, ACM 162

[33] Kuźnicka E and Gburzyński P 2017 *Automatic detection of suckling events in lamb through accelerometer data classification*, *Computers and Electronics in Agriculture* **138** 137

[34] Gburzynski P, Kaminska B and Rahman A 2009 *On reliable transmission of data over simple wireless channels*, *Journal of Computer Systems, Networks and Communications*

[35] Ma S, Yang Y, Qian Y, Sharif H and Alahmad M 2016 *Energy harvesting for wireless sensor networks: applications and challenges in smart grid*, *International Journal of Sensor Networks* **21** (4) 226

[36] Adu-Manu K S, Adam N, Tapparello C, Ayatollahi H and Heinzelman W 2018 *Energy-harvesting wireless sensor networks (EH-WSNs) a review*, *ACM Transactions on Sensor Networks (TOSN)* **14** (2) 1

[37] Ali A, Jadoon Y K, Changazi S A and Qasim M 2020 *Military operations: Wireless sensor networks based applications to reinforce future battlefield command system*, *2020 IEEE 23rd International Multitopic Conference (INMIC)*, IEEE 1

[38] Clemente A D S B, Martínez-de Dios J D and Baturone A O 2012 *A WSN-based tool for urban and industrial fire-fighting*, *Sensors* **12** (11) 15009

[39] Levis P 2009 *TinyOS Programming*, Cambridge University Press

[40] Texas Instruments *TI-RTOS 2.20 User's Guide*, *2016*, Technical document SPRUHD4M

[41] Dunkels A, Gronvall B and Voigt T 2004 *Contiki-a lightweight and flexible operating system for tiny networked sensors*, *29th annual IEEE international conference on local computer networks*, IEEE 455

[42] Levis P, Lee N, Welsh M and Culler D 2003 *TOSSIM: Accurate and scalable simulation of entire tinyos applications*, *Proceedings of the 1st international conference on Embedded networked sensor systems* 126

[43] Zikria Y B, Afzal M K, Ishmanov F, Kim S W and Yu H 2018 *A survey on routing protocols supported by the Contiki Internet of Things operating system*, *Future Generation Computer Systems* **82** 200

[44] Sobeih A, Hou J C, Kung L-C, Li N, Zhang H, Chen W-P, Tyan H-Y and Lim H 2006 *J-Sim: a simulation and emulation environment for wireless sensor networks*, *IEEE Wireless Communications* **13** (4) 104

[45] Wu H, Luo Q, Zheng P and Ni L M 2007 *VMNet: Realistic emulation of wireless sensor networks*, *IEEE Transactions on Parallel and Distributed Systems* **18** (2) 277

[46] Li J and Serpen G 2012 *Simulating heterogeneous and larger-scale wireless sensor networks with TOSSIM TinyOS emulator*, *Procedia Computer Science* **12** 374

[47] Riliskis L and Osipov E 2013 *Symphony: Simulation, emulation and virtualization framework for accurate WSN experimentation*, *2013 4th International Workshop on Software Engineering for Sensor Network Applications (SESENA)* 1

[48] Manatakis D V, Nennes M G, Bakas I G and Manolakos E S 2014 *Simulation-driven emulation of collaborative algorithms to assess their requirements for a large-scale WSN implementation*, *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, IEEE 8360

[49] Olsonet Communications Corporation 2019 *Programming under PicOS*, *PicOS Documentation*

[50] Akhmetshina E, Gburzyński P and Vizeacoumar F 2003 *PicOS: A tiny operating system for extremely small embedded platforms*, *Proceedings of ESA'03 (Las Vegas)* 116

[51] Shimony B, Nikolaidis I, Gburzynski P and Stroulia E 2010 *PicOS tuples: easing event based programming in tiny pervasive systems*, *Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, ACM 53

[52] Boers N M, Gburzyński P, Nikolaidis I and Olesiński W 2010 *Developing wireless sensor network applications in a virtual environment*, *Telecommunication Systems* **45** (2-3) 165

[53] Levis P et al. 2005 *TinyOS: An operating system for sensor networks*, *Ambient Intelligence*, Springer 115

[54] Regehr J, Reid A and Webb K 2005 *Eliminating stack overflow by abstract interpretation*, *ACM Transactions on Embedded Computing Systems (TECS)* **4** (4) 751

[55] Bucur D 2012 *Temporal monitors for TinyOS*, *International Conference on Runtime Verification*, Springer 96

[56] Moura A L D and Ierusalimschy R 2009 *Revisiting coroutines*, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **31** (2) 1

[57] Hambarde P, Varma R and Jha S 2014 *The survey of real time operating system: RTOS*, *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*, IEEE 34

[58] Gburzynski P and Kaminska B 2008 *Testing real-time properties of embedded systems*, *Proceedings of ESA 2008* 179

[59] Gburzyński P 2019 *Modeling Communication Networks and Protocols*, Springer

[60] Dobosiewicz W and Gburzyński P 1997 *Protocol design in SMURPH*, *State of the art in Performance Modeling and Simulation*, Gordon and Breach 255

[61] Gburzyński P and Nikolaidis I 2006 *Wireless network simulation extensions in SMURPH /SIDE*, *Proceedings of the 2006 Winter Simulation Conference (WSC'06)*

[62] Gburzyński P and Maitan J 1997 *Simulation and control of reactive systems*, *Proceedings of Winter Simulation Conference WSC'97* 413

[63] Gburzyński P, Maitan J and Hillyer L 1998 *Virtual prototyping of reactive systems in SIDE*, *Proceedings of the 5th European Concurrent Engineering Conference ECEC'98* 75

[64] Gburzyński P 1996 *Protocol design for local and metropolitan area networks*, Prentice-Hall

[65] Obaidat M S, Zarai F and Nicopolitidis P 2015 *Modeling and simulation of computer networks and systems: Methodologies and applications*, Morgan Kaufmann

[66] Barrenetxea G, Ingelrest F, Schaefer G and Vetterli M 2008 *The hitchhiker's guide to successful wireless sensor network deployments*, *Proceedings of the 6th ACM conference on Embedded network sensor systems* 43

[67] Liu A, Zheng Z, Zhang C, Chen Z and Shen X 2012 *Secure and energy-efficient disjoint multipath routing for WSNs*, *IEEE Transactions on Vehicular Technology* **61** (7) 3255

[68] De Couto D S, Aguayo D, Chambers B A and Morris R 2003 *Performance of multihop wireless networks: Shortest path is not enough*, *ACM SIGCOMM Computer Communication Review* **33** (1) 83

[69] Gburzyński P, Kaminska B and Olesiński W 2007 *A tiny and efficient wireless ad-hoc protocol for low-cost sensor networks*, *Proceedings of DATE'07* 1562

[70] Rahman A, Olesiński W and Gburzyński P 2004 *Controlled flooding in wireless ad-hoc networks*, *Proceedings of IWWAN'04*

[71] Shakkottai S, Rappaport T and Karlsson P 2003 *Cross-layer design for wireless networks*, *IEEE Communications Magazine* **41** (10) 74

[72] Akyildiz I F and Wang X 2008 *Cross-layer design in wireless mesh networks*, *IEEE Transactions on Vehicular Technology* **57** (2) 1061

[73] Feng Y, Zhang B, Chai S, Cui L and Li Q 2017 *An optimized AODV protocol based on clustering for WSNs*, *6th International Conference on Computer Science and Network Technology (ICCSNT)*, IEEE 410

[74] Chatzimisios P, Boucouvalas A and Vitsas V 2004 *Effectiveness of RTS/CTS handshake in IEEE 802.11 a wireless LAN*, *Electronics Letters* **40** (14) 915

[75] Fahmy H M A 2021 *Protocol stack of WSNs*, *Concepts, Applications, Experimentation and Analysis of Wireless Sensor Networks*, Springer 53

[76] Ray S, Carruthers J B and Starobinski D 2003 *RTS/CTS-induced congestion in ad hoc wireless LANs*, *2003 IEEE Wireless Communications and Networking*, IEEE, **3** 1516

[77] Rahman A and Gburzyński P 2006 *Hidden problems with the hidden node problem*, *Proceedings of 23rd Biennial Symposium on Communications* 270

[78] Wang L, Wu K and Hamdi M 2012 *Combating hidden and exposed terminal problems in wireless networks*, *IEEE Transactions on Wireless Communications* **11** (11) 4204

[79] Hassan W H et al. 2019 *Current research on internet of things (IoT) security: A survey*, *Computer Networks* **148** 283

[80] Perrig A, Szewczyk R, Tygar J D, Wen V and Culler D E 2002 *SPINS: Security protocols for sensor networks*, *Wireless networks* **8** (5) 521

[81] Mbarek B and Meddeb A 2016 *Energy efficient security protocols for wireless sensor networks: SPINS vs TinySec*, *2016 International Symposium on Networks, Computers and Communications (ISNCC)*, IEEE 1

[82] Khan M A and Salah K 2018 *IoT security: Review, blockchain solutions and open challenges*, *Future Generation Computer Systems* **82** 395

[83] Gay D, Levis P, Von Behren R, Welsh M, Brewer E and Culler D 2014 *The nesc language: A holistic approach to networked embedded systems*, *Sigplan Notices* **49** (4) 41

[84] Daemen J and Rijmen V 2013 *The design of Rijndael: AES-the advanced encryption standard*, Springer Science & Business Media

[85] Rogaway P, Wooding M and Zhang H 201 *The security of ciphertext stealing*, *International Workshop on Fast Software Encryption*, Springer 180

[86] Bellare M, Kilian J and Rogaway P 2000 *The security of the cipher block chaining message authentication code*, *Journal of Computer and System Sciences* **61** (3) 362

[87] Raymond D R and Midkiff S F 2008 *Denial-of-service in wireless sensor networks: Attacks and defenses*, *IEEE Pervasive Computing*, **1** 74

[88] Seba A, Nouali-Taboudjemat N, Badache N and Seba H 2019 *A review on security challenges of wireless communications in disaster emergency response and crisis management situations*, *Journal of Network and Computer Applications* **126** 150

[89] Ghildiyal S, Mishra A K, Gupta A and Garg N 2014 *Analysis of denial of service (DoS) attacks in wireless sensor networks*, *IJRET: International Journal of Research in Engineering and Technology* **3** 2319

Paweł Gburzyński received his MSc and PhD in Computer Science from the University of Warsaw, Poland in 1976 and 1982, respectively. Between 1984 and 2010 he was with the Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada from where he retired in 2010 as full professor. Since 2014 he has been a professor at the Faculty of Art, Technology and Communication at Vistula University in Warsaw, Poland. Dr. Gburzynski has worked extensively with the industry being involved in numerous projects with companies in Canada and the United States. His research interests are in telecommunication, embedded systems, operating systems, simulation, and performance evaluation. In 2002 he co-founded Olsonet Communications, a small company incorporated in Canada designing and manufacturing custom wireless sensing systems.