Paper

On a practical approach to low-cost ad hoc wireless networking

Paweł Gburzyński and Włodzimierz Olesiński

Abstract—Although simple wireless communication involving nodes built of microcontrollers and radio devices from the low end of the price spectrum is quite popular these days, one seldom hears about serious wireless networks built from such devices. Most of the commercially available nodes for ad hoc networking (somewhat inappropriately called "motes") are in fact quite serious computers with megabytes of RAM and rather extravagant resource demands. We show how one can build practical ad hoc networks using the smallest and cheapest devices available today. In our networks, such devices are capable of sustaining swarm-intelligent sophisticated routing while offering enough processing power to cater to complex applications involving distributed sensing and monitoring.

Keywords— ad hoc wireless networks, sensor networks, operating systems, reactive systems, specification, simulation.

1. Introduction

The vast number of academic contributions to ad hoc wireless networking have left a surprisingly tiny footprint in the practical world. For once, the industry is not much smarter, although for a different reason. While the primary problem with academic research, not only in this particular area, is its excessive separation from mundane and academically uninteresting aspects of reality, the industry appears to suffer from its inherent inability to think small and holistic. The net outcome is in fact the same in both cases: the popular and acknowledged routing schemes, as well as programmer-friendly application development systems, require a significant amount of computing power and are not suitable for small and inexpensive devices. As an example of the latter, consider a key-chain car opener. A networking "node" of this kind is typically built around a low-power microcontroller with ~ 1 KB of RAM driving a simple transceiver. The combined cost of the two components is usually below $5. While it is not a big challenge to implement within this framework a functionally simple broadcaster of short packets, it is quite another issue to turn this device into a collaborating node of a serious ad hoc wireless system.

The plethora of popular ad hoc routing schemes proposed and analyzed in the literature [6, 22, 25, 27, 31, 32, 33, 38], addresses devices with a somewhat larger resource base. This is because those schemes require the nodes to store and analyze a non-trivial amount of information to carry out their duties. Moreover, none of them provides for "graceful downscaling," whereby a node with a smaller than "recommended" amount of RAM can still fulfill its obligation to the network, perhaps at a reduced level. With such systems, hardware resources must be overdesigned (i.e., wasted in typical scenarios), as an overrun leads to a functional breakdown.

The most popular commercial scheme originally intended for building ad hoc networks is Bluetooth. Its two fundamental problems are:

- a large footprint and, consequently, non-trivial cost and power requirements;
- arcane connection discovery and maintenance options, which render true ad hoc networking cumbersome.

Even though some attempts are still being made to build actual ad hoc networks based on Bluetooth [18], it is commonly agreed that the role of this technology is reduced to creating small personal-area hubs. ZigBee® (based on ad hoc on-demand distance vector (AODV) [34]) comes closer; however, despite the tremendous industrial push, it fails to catch on. The reason, we believe, is its isolation from the wider context of application development issues combined with the limited flexibility of AODV as a routing scheme.

If there is a place in the realm of low-end microcontrollers where the adjective "ad hoc" is well applicable, it is to software development. Typically, the software (firmware) designed for one particular project is viewed as a "one-night stand", and its re-usability, modularity, and exchangeability are not deemed interesting. This is because convenient, modular, and self-documentable programming techniques based on concepts like multi-threading, event handling, synchronization, object-orientedness are considered too costly in terms of resource requirements (mostly RAM) to be of merit in programming the smallest microcontrollers. Even TinyOS [26], which is the most popular operating system for networked microcontrollers, has many shortcomings in these areas. Moreover, its evolution (as it usually happens with systems driven by large communities and consortia), leans towards larger and larger devices.

Serious efforts to introduce an order and methodology into programming small devices often meet with skepticism and shrugs. The common misconception is that one will not have to wait for long before those devices disappear and become superseded by larger ones, capable of running Linux or Windows®. This is not true. Despite the ever decreasing cost of microcontrollers, we see absolutely no reduction in the demand for the ones from the lowest end of the spectrum. On the contrary: their low cost and low power re-

quirements enable new applications and trigger even more demand.

One of our claims is that the only way to harness trivially small microcontrolled devices to performing complex communal tasks, amounting to effective and efficient ad hoc networking, is to follow a holistic approach to organizing their software. This is in fact the primary problem with the industry: their approach to solutions is *layered*, in the sense that separable components of the traget product are viewed by them as isolated subproblems to be attacked by different teams equipped with different tools and driven by different objectives. Consider a node of a wireless ad hoc network with the components listed in Fig. 1. In a typical production cycle, each of those components is viewed as an end in itself. From our perspective as academics, we tend to focus on the *protocols* component, forgetting that it is merely a fragment of something that may ever be useful. As it happens, the most mundane fragment of the whole picture, i.e., the hardware, directly determines the viability of the entire project as a commercial idea.

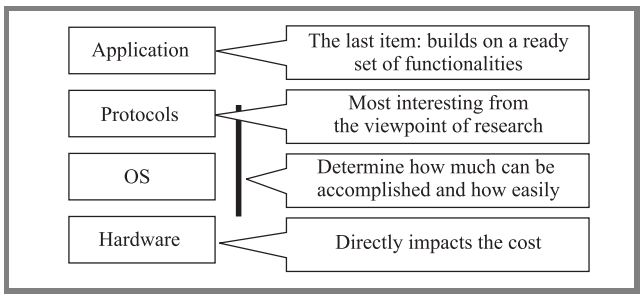

Fig. 1. Product "layers" of a wireless node.

A natural industrial reflex when it comes to protocols is standardization. While some of its benefits are unquestionable, e.g., the interoperability of diverse equipment, the main thrust of the standardization efforts that we in fact see in the area of low-cost wireless networking is aimed at the "soft" parts of the protocols (above the physical layer). Note that this has brought us ZigBee, which tries to impose on us ready network-layer paradigms for implementing operations as delicate as forwarding within unknown networks catering to unknown applications. Notably, in the very area where the standardization would be truly and indisputably useful, i.e., in the physical layer, we saw a complete lack of interest from the manufacturers, at least in the pre-ZigBee era. A stress on interoperability at that stage would have been considerably more beneficial to the community of users and developers of ad hoc wireless communication solutions. These days, it only happens in the context of ZigBee, which many of the manufacturers come to perceive as a curse of their membership in the consortium. Thus, they usually provide ways to bypass the ZigBee stack and enable various clumsy (but at least partially feasible) ad hoc networking scenarios, e.g., hubs or limited multi-hopping.

By their very nature, standards devised in isolation from the view of applications are bound to result in oversized footprints. This is because:

- They have to anticipate many circumstances that will occur marginally, if ever, in any particular application.
- They are devised by committees consisting of people with conflicting ideas and agendas, and tend to accommodate a little bit of everything – to satisfy all members.
- Their designers focus on functionality rather than feasibility: the lack of a reference point (application, hardware) makes it difficult to see the complexity of implementation.

In this paper, we outline our comprehensive platform for rapidly building wireless *praxes*, i.e., low-cost applications based on ad hoc networking. This platform comprises an operating system, a flexible, layer-less, and auto-scalable ad hoc forwarding scheme, and an emulator for testing the praxes in a high-fidelity virtual environment. We show how one can build well structured multithreaded programs operating within a trivially small amount of RAM and organize them into authentic ad hoc wireless applications.

2. The operating system

The foundation of our development platform is PicOS: a tiny operating system for small-footprint microcontrolled devices executing reactive applications [1, 39].

2.1. A historical perspective

The ideas that have found their way into PicOS originated as early as 1986. About that time, many published performance studies of carrier sense multiple access/collision detection (CSMA/CD)-based networks (like Ethernet) had been subjected to heavy criticism from the more practically inclined members of the community – for their irrelevance and overly pessimistic conclusions. The culprit, or rather culprits, were identified among the popular collection of models (both analytical and simulation), whose cavalier application to describing poorly understood and crudely approximated phenomena had resulted in worthless numbers and exaggerated blanket claims [5]. Our own studies of low-level protocols for local-area networks, on which we embarked at that time [8–12, 14–16], were aimed at devising novel solutions, as well as dispelling myths surrounding the old ones. Owing to the fact that exact analytical models of the interesting systems were (and still are) nowhere in sight, the performance evaluation component of our work relied heavily on simulation. To that end, we developed a detailed network simulator, called LANSF (local area network simulation facility) and its successor SMURPH (sys-

tem for modeling unslotted real-time phenomena) [13, 19], which carefully accounted for all the relevant physical phenomena affecting the correctness and performance of low-level protocols, e.g., the finite propagation speed of signals, race conditions, imperfect synchronization of clocks, variable event latency incurred by realistic hardware. In addition to numerous tools facilitating performance studies, SMURPH was also equipped with instruments for dynamic conformance testing [3].

At some point we couldn't help noticing that the close-to-implementation appearance of SMURPH models went beyond mere simulation: the same paradigm could be used for implementing certain types of real-life applications. The first outcome of that observation was an extension of SMURPH into a programming platform for building distributed controllers of physical equipment represented by collections of sensors and actuators. Under its new name, SIDE (sensors in a distributed environment), the package encompassed the old simulator augmented by tools for interfacing its programs to real-life objects [20, 21].

A natural next step was to build a complete and self-sustained executable platform (i.e., an operating system) based entirely on SMURPH. It was directly inspired by a practical project whose objective was to develop a low-cost intelligent badge equipped with a low-bandwidth, short-range, wireless transceiver allowing it to communicate with neighbors. As most of the complexity of the device's behavior was in the communication protocol, its model was implemented and verified in SMURPH. The source code of the model, along with its plain-language description, was then sent to the manufacturer for a physical implementation. Some time later, the manufacturer sent us back their prototype microprogram for "optical" conformance assessment. Striving to fit the program's resources into as little memory (RAM) as possible, the implementer organized it as an extremely messy single thread for the bare CPU. The program tried to approximate the behavior of our high-level multi-threaded model via an unintelligible combination of flags, hardware timers and counters. Its original, clear, and self-documenting structure, consisting of a handful of simple threads presented as finite state machines, had completely disappeared in the process of implementation. While struggling to comprehend the implementation, we designed a tiny operating system providing for an easy, natural and rigorous implementation of SMURPH models on microcontrollers. Even to our surprise, we were able to actually *reduce* the RAM requirements of the re-programmed application. Needless to say, the implementation became clean and clear: its verification was immediate.

2.2. PicOS threads

The most serious problem with implementing non-trivial, structured, multitasking software on microcontrollers with limited RAM is minimizing the amount of memory resources needed to describe a thread. While the basic record of a thread in the kernel of an embedded system can be contained in a handful of simple variables (status, code pointer, data pointer, one or two links), the most troublesome component of the thread footprint is its stack, which must be preallocated to every thread in a safe amount sufficient for its maximum possible need. In addition to providing room for the automatic variables used by thread functions, including the implicit ones (like return addresses), the stack is an important part of the thread's context. When the thread is preempted, the stack preserves the snapshot of its trace, which will make it possible to resume the thread later, in a consistent and transparent manner.

At first sight, it might seem that microcontrollers with very small amount of RAM are condemned to running threadless systems. For example, in TinyOS [24, 26], the issue of limited stack space has been addressed in a radical manner – by avoiding multithreading altogether. Essentially, TinyOS defines two types of activities: event handlers (corresponding to interrupt service routines and callbacks) and the so-called *tasks*, which are simply chunks of code that cannot be preempted by (and thus cannot dynamically coexist with) other tasks.

One way to strike a compromise between the complete lack of threads on the one hand, and overtaxing the tiny amount of RAM with partitioned and fragmented stack space on the other, may be to reduce the flexibility of threads regarding the circumstances under which they can be preempted (i.e., lose the CPU). The idea is to create an environment where the thread is forced to relinquish its stack before preemption. That would restrict the preemption opportunities to a collection of *checkpoints* of which the thread would be aware. By stimulating a structured organization of those checkpoints, we could try to

- avoid locking the CPU at a single thread for an extensive amount of time;
- turn them into natural and useful elements of the thread's specification, e.g., enhancing its clarity and reducing the complexity of its structure.

These ideas lie at the heart of PicOS's concept of threads, which are structured like finite state machines (FSM) and exhibit the dynamics of coroutines [4, 7] with multiple entry points and implicit control transfer.

For illustration, consider the sample thread code shown in Fig. 2. This is in fact a C function: any exotic keywords or constructs are straightforward macros handled by the standard C preprocessor. The states are marked by the `entry` statements. Whenever a thread is assigned the CPU, its code function is invoked in the *current state*, i.e., at one specific entry point.

State boundaries represent the checkpoints at which a thread can be preempted and resumed. The way it works is that a thread can only lose the CPU when it explicitly relinquishes control at the boundary of its current state. In particular, this happens when the thread executes `release`, as within state `RC_PASS` in Fig. 2. This has the effect of returning the CPU to the scheduler, which is then free

to allocate it to another thread. A thread receiving the CPU will always find itself at the entry point to one of its states.

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

Fig. 2. A sample thread code in PicOS.

Typically, before executing `release`, a thread issues a number of *wait requests* specifying one or more conditions (events) to resume it in the future. Then, the effect of `release` is to block the thread until at least one of the conditions is fulfilled. If multiple events are awaited by the thread, the earliest of them will wake it up. Once that happens, all the pending wait requests are erased: the thread has to specify them from scratch at every wake-up. As a wait request, besides the condition, specifies the state to be assumed by the awakened thread, the collection of wait requests issued by a thread in every state describes the options for its transition function from that state.

In state `RC_PASS` (Fig. 2), if the `if` condition holds, the thread issues two wait requests: one with `when` and the other with `delay`. With `when`, the thread declares that it wants to be resumed in state `RC_PASS` upon the occurrence of an event represented by the address of a data object (`buffer`). Such events can be signaled with `trigger`, as illustrated in state `RC_ENP`. The `delay` operation sets up an alarm clock for the prescribed number of milliseconds (1000). The event waking the process up will be triggered when the alarm clock goes off.

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

The above paradigm of organizing tasks in PicOS has proved very friendly, versatile, and useful for describing the kinds of applications typical of embedded systems, i.e., reactive ones [1, 39]. The FSM layout of the praxis comes for free and can be mechanically transformed, e.g., into a statechart [17, 23], for easy comprehension or verification. Owing to the fact that a blocked thread needs no stack space, all threads in the system can share the same single global stack. The programmer-controlled preemptibility grain practically eliminates all synchronization problems haunting traditional multi-threaded applications.

2.3. The footprint

So far, PicOS has been implemented on the MSP430 microcontroller family and on eCOG1 from Cyan Technology. A port to ARM7 is under way. The size of the thread control block (TCB) needed to describe a single PicOS thread is adjustable by a configuration parameter, depending on the number of events E that a single thread may want to await simultaneously. The standard setting of this number is 3, which is sufficient for all our present applications and protocols. The TCB size in bytes is equal to $8+4E$, which yields 20 bytes for $E = 3$. The number of threads in the system (the degree of multiprogramming) has no impact on the required stack size, which is solely determined by the maximum configuration of nested function calls. As automatic variables are not very useful for threads (they do not survive state transitions and are thus discouraged), the stack has no tendency to run away. 96 bytes of stack size is practically always sufficient. In many cases, this number can be reduced by half.

2.4. System calls

In a traditional operating system, a thread may become blocked implicitly when it executes a system call that cannot complete immediately. To make this work with PicOS threads, which can only be blocked at state boundaries, potentially blocking system calls must incorporate a mechanism involving a combination of a wait request and `release`. For illustration, see the first statement in state `RC_TRY` (Fig. 2). Function `tcv_rnp` (belonging to VNETI – see Subsection 2.5) is called to receive a packet from a network session represented by descriptor `efd`. It may return immediately (if a packet is already available in the buffer), or block (if the packet is yet to arrive). In the latter case, the system call will block the thread and resume it in the indicated state when it makes sense to re-execute `tcv_rnp`, i.e., upon a packet reception.

Essentially, there are two categories of system calls in PicOS that may involve blocking. The first one, like `tcv_rnp`, may get into a situation when something needed by the program is not immediately available. Then, the event waking up the process will indicate a new acquisition opportunity: the failed operation has to be re-done. The second scenario involves a delayed action that must be internally completed by the system call before the process becomes runnable again. In such a case, the event indicates that the process may continue: it does not have to re-execute the system call. To keep the situation clear, the syntax of system call functions unambiguously determines which is the case. Namely, for the first type of calls,

the state argument is first on the argument list, while it is last for the second type. Incidentally, all system calls that can ever block take more than one argument.

2.5. The versatile network interface (VNETI)

The interface of a PicOS application (praxis) to the outside world is governed by a powerful module called VNETI. VNETI offers to the praxis a simple and orthogonal collection of application programming interface (API), independent of the underlying implementation of networking. To avoid the protocol layering problems haunting small-footprint solutions, VNETI is completely layer-less and its semi-complete generic functionality is redefined by plug-ins.

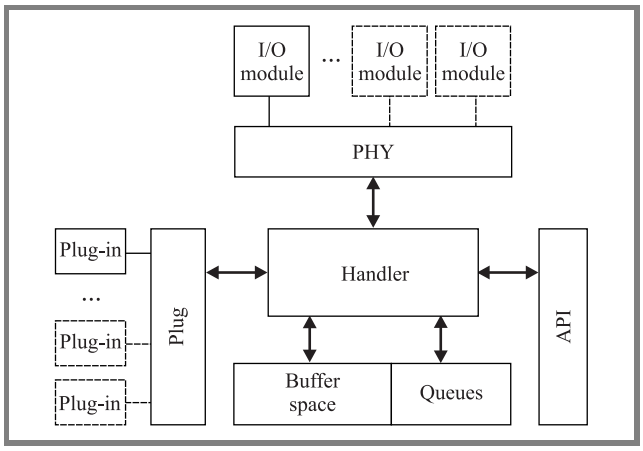

Fig. 3. The structure of VNETI.

The structure of VNETI is shown in Fig. 3. Standard sets of interfaces are provided for attaching drivers of physical communication modules (PHY), as well the plug-ins representing the *protocols* configured into the system. The API available to the praxis consists of a fixed set of operations that are independent of the configured assortment of plug-ins or the physical modules.

```
typedef struct {
  int (*tcv_ope) (int phid, int fd, va_list ap);
  int (*tcv_clo) (int phid, int fd);
  int (*tcv_rcv) (int phid, address buf, int len, int *ses,
                                                tcvadp_t *frm);
  int (*tcv_frm) (address packet, int phid, tcvadp_t *frm);
  int (*tcv_out) (address packet);
  int (*tcv_xmt) (address packet);
  int (*tcv_tmt) (address packet);
  int tcv_info;
} tcvplug_t;
```

Fig. 4. Plug-in interface.

A plug-in is described by a numerical identifier and a set of operations, as shown in Fig. 4. Generally, those operations intercept various requests coming from the praxis, as well as the packets, as they make their passes through the buffer storage of VNETI (Fig. 3). For example, when the praxis opens a communication session, by executing the `tcv_open` function of VNETI, it specifies the identity of the plug-in to be responsible for the session. Thus, VNETI will invoke the *ope* (`tcv_ope`) function provided by the plug-in, to carry out any specific administrative operations required to set up the session. Now, consider a packet being received from the network. The PHY module receiving the packet presents it to VNETI by invoking a standard interface function. In response, VNETI will in turn apply to the packet the *rcv* functions (`tcv_rcv` in Fig. 4) of all the configured plug-ins. Based on the packet content and the identity of the PHY module, the function may decide to *claim* the packet (by returning a special value) and assign it to a particular session (the return argument `ses`), which typically corresponds to one of the active session being handled by the praxis and associated with the plug-in. The last argument of `tcv_rcv` returns the pointers to the packet's low-level header and trailer, which will be discarded when the claimed packet is deposited by VNETI in its buffer space.

The set od operations available to plug-ins involve queue manipulations, cloning packets, inserting special packets, and assigning to them the so-called *disposition codes* representing various processing stages. Any sophisticated protocol (e.g., TCP/IP) can be implemented within this paradigm. Its underlying premise is to treat all packets "holistically" with no regard for any assumed *layers* of their processing.

2.6. Real time considerations

One problem resulting from the limited preemptibility of threads is its potentially detrimental impact on the real-time behavior of the embedded application [2, 29]. This is because the maximum rescheduling time for any thread (regardless of its priority [37]) will include the maximum non-preemptibility interval for any other thread in the system.

PicOS scheduler admits several options, which can be selected at the time the system is compiled into a praxis. The most naive (and also the most popular) of those options implements a fixed priority (non-preemptive) scheme, whereby the threads are sorted in the decreasing order of their importance. Whenever a thread releases the CPU, the scheduler assigns it to the first thread that is not waiting for any event. This way, when multiple threads are ready to run, the one closer to the front of the list will win the CPU.

In most cases, this trivial approach to scheduling is quite adequate to fulfill the real time requirements of the application, especially if those requirements are soft. This is because reactive applications tend to do little computations (are not CPU bound) and focus on processing events, which actions typically take a small amount of time. Notably, the truly critical actions (like extracting data from time-constrained peripheral equipment) are carried out in interrupts, which are not subjected to the kind of postponement exercised by threads. Consequently,

the limited preemptibility of the latter does not impair the rate at which external events can be formally absorbed by the PicOS application. The way interrupt service routines are organized in PicOS renders them interruptible: consequently an interrupt service routine can be preempted by another interrupt service routine, according to the pre-declared criteria of importance. To avoid inflating the bound on the maximum stack size, this feature is optional: it can be turned on selectively, on a per-interrupt basis, to cater to the hard real-time requirements of critical peripherals.

It is a good practice to organize PicOS threads in such a way as to keep the maximum execution time of a single state reasonably small. Note that the granularity of thread states is under the programmer's control. There is virtually no penalty for introducing extra states with the purpose of bringing down the maximum duration of a CPU burst exhibited by the thread. In many circumstances, in addition to improving the real-time behavior of the entire application, this approach enhances the clarity, self-documentability, and re-usability of the thread code.

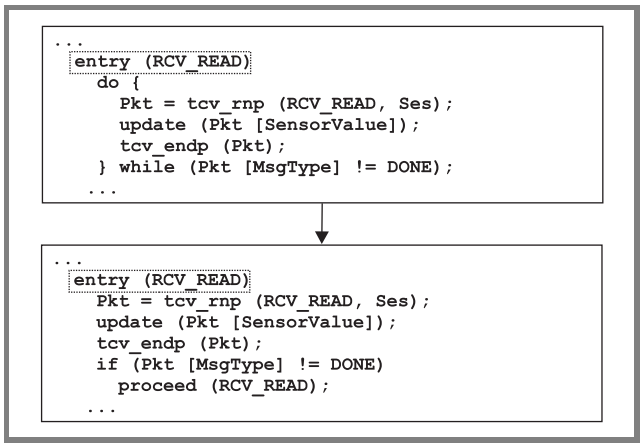

Fig. 5. Avoiding non-trivial loops in PicOS threads.

It is also recommended to avoid non-trivial loops that do not cross state boundaries. Such loops can be always converted as shown in Fig. 5, i.e., by starting the loop at a state boundary and closing it with `proceed`, which has the effect of enabling preemptibility at every turn. Although `proceed` has the appearance of "goto," the operation involves an *actual* state transition, i.e., the thread function is exited and re-entered via the scheduler, which is thus allowed to interleave other threads with the loop. This way, any higher priority threads will be able to claim their share of the CPU in between the loop turns.

All internal functions (system calls) of PicOS have been programmed with consistent adherence to the principle of simplicity and orthogonality. This also applies to the internals of VNETI. Every function implements a loop-less action, whose execution time is approximately constant. This way, the timing of code referencing such functions is easy to estimate within a very narrow uncertainty margin. Consequently, it is possible to carry out meaningful real-time assessments, including hard real-time guarantees, by estimating the execution time of thread states. The latter can be often accomplished by a purely mechanical analysis of the compiler output, i.e., the tally of the CPU cycles in a loop-less sequence of machine instructions.

3. Communication

Most routing protocols for ad hoc wireless networks, as described in the literature, assume point-to-point communication, whereby each node forwarding the packet on its way to the destination sends it to a specific neighbor. Regardless of whether the scheme is proactive [6, 33] or reactive [22, 25, 27, 31, 32, 38], its primary objective is to determine the exact sequence of nodes to forward the packets from point *A* to point *B*. Despite the fact that the wireless environment is inherently broadcast, this free feature is rarely exploited during the actual forwarding of session packets, although all protocols necessarily take advantage of broadcast transmissions during various stages of route discovery (e.g., the periodic HELLO messages broadcast by all nodes to announce their presence in the neighborhood). For example, in AODV [34], a node *S* initiating packet exchange with node *D* broadcasts a request to its one-hop neighbors to start the so-called path discovery operation. Based on its current perception of the neighborhood and cached information collected form previous path discoveries, a node receiving such a request may decide to forward it elsewhere, or respond with a path information intended for the initiating node *S*. At the end, a single path between *S* and *D* has been established. A problem arises when the path is broken, because such a mishap effectively demolishes the entire delicate structure. When that happens, a new path discovery operation is essentially started from scratch.

On top of the susceptibility to node failures and disappearance (mobility), this generic approach requires the nodes to store a potentially sizable amount of elaborate routing information, which cannot be made fuzzy. For example, if a node is unable to store the identity of the next-hop neighbor for a particular session, then it will simply not be able to carry out its duties with respect to that session (thus breaking the path). It may confuse the network by offering a service that it is unable to deliver, resulting in stalled path discovery and, ultimately, communication failure.

3.1. Tiny ad hoc routing protocol (TARP): forwarding by re-casting

The idea behind our solution, dubbed TARP, is to embrace fuzziness as a useful feature and take full advantage of the inherently broadcast nature of the wireless medium. Traditional schemes view this nature as a rather serious problem and try to defeat its negative consequences (hidden/exposed terminals) via MAC-level handshakes intended to facilitate point-to-point transmission [28]. TARP, in contrast, turns it into an advantage.

Suppose that node *S* wants to send a packet to node *D*. With TARP, *S* simply transmits (broadcasts) the packet to its neighbors. A neighbor may decide to drop the packet (if it believes that its contribution to the communal forwarding task will not help) or retransmit it. This process continues until the packet reaches the destination *D*. An important property of this generic scheme (otherwise known as flooding) is that a retransmitted packet is never specifically addressed to a single next-hop neighbor. Needless to say, to make it useful, measures must be taken to limit the number of retransmissions to the minimum at which the desired quality of service is maintained. This part comes as a series of rules that determine when a node receiving a packet should rebroadcast it, as opposed to dropping. Some ideas for such rules are obvious, e.g., discarding duplicates of already seen packets and limiting the maximum number of hops that a packet can travel.

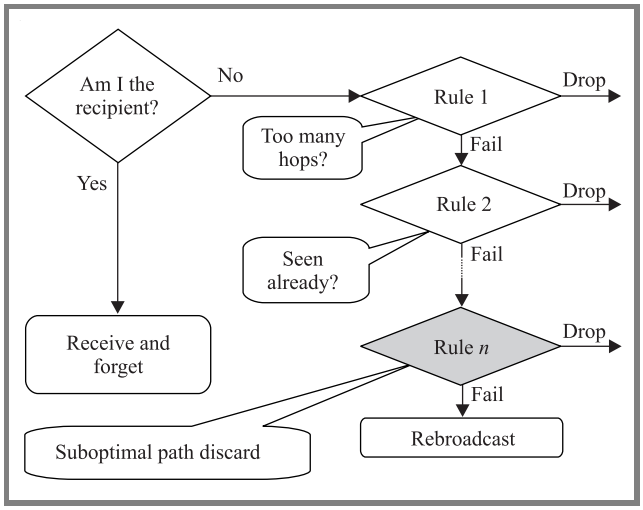

Fig. 6. Application of rules in TARP.

Figure 6 illustrates the way TARP applies its rules to an incoming packet. The important property of *any* rule implanted into TARP is its naturally conservative behavior in the face of incomplete information (uncertainty). This means that a rule that does not know what to do always *fails*, which is to say that the packet will not be dropped on its account. As most of the rules are cache driven, such conservative behavior provides for automatic scalability of the rule to the limitations of its resources. A better-equipped rule may tend to drop more packets and thus avoid polluting the neighborhood with superfluous traffic. The same rule executed on a device with smaller memory may not be as exacting, but if it errs, it does so on the safe side, i.e., it drops no packets that would not have been dropped, had the node been more resourceful. This is something that point-to-point forwarding protocols find difficult to accomplish. To them, a path is just a path: you either know the precise identity of your next hop neighbor, or you know nothing at all. There is no room for fuzziness in that kind of setup.

3.2. The selective packet discard (SPD) rule

The key to the success of our variant of flooding is the most representative rule of TARP, one that brings the paths traveled by forwarded packets down to a narrow (but intentionally fuzzy) stripe of nodes along the shortest route. This rule is named SPD, for suboptimal path discard.

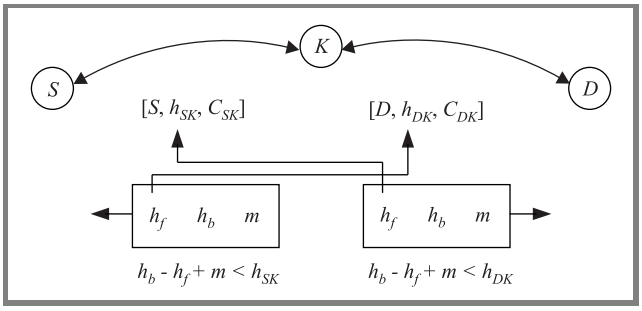

Fig. 7. The rule for selective packet discard.

Consider the three nodes shown in Fig. 7. *K* is contemplating whether it should re-broadcast an "overheard" packet sent by *S* and addressed to *D*. Suppose *K* knows this information: h_b – the total number of hops traveled by some packet that has recently reached *S* arriving from *D* (in the opposite direction); h_f – the number of hops traveled so far by the current packet; h_{DK} – the number of hops separating *K* from *D*. If $h_b < h_f + h_{DK}$, *K* can suspect that the packet can make it to *D* via a shorter path leading through another node. This is because, apparently, packets can make it from *D* to *S* in fewer hops than the combination of whatever the packet has already gone through with the number of hops it still must cover if forwarded via *K*. Thus, in such a case, *K* may decide to drop the packet.

The requisite information can be collected from the headers of packets that *K* overhears as the session goes on. To make it possible, the packet header should carry the current number of hops traveled by the packet as well as the number of hops traveled by the last packet that arrived at the sender from the opposite end. As a duplicate packet is always discarded at the earliest detection, a non-duplicate packet arriving at a destination makes it along the fastest (and usually the shortest) path. Until the network learns about a particular session (understood as a pair of nodes that want to communicate), the forwarding for that session may be overly redundant.

Owing to the inherent imperfections of the ad hoc wireless environment, *K* should not be too jumpy with negative decisions. TARP uses two adjustable ways to damp the behavior of the SPD rule to account for the uncertainty of knowledge. One of them is the slack parameter *m* shown in the inequality in Fig. 7. When $m > 0$, the rule will allow the node to forward the packet even though the path that it is able to offer appears to be slightly longer (by up to *m* hops) than the currently believed shortest path.

Each entry in the SPD cache, in addition to the node identifier and the current estimate for the number of hops, carries

a counter (C_{SK} and C_{DK} in Fig. 7), which is incremented by 1 each time the rule succeeds on that entry (i.e., the packet is dropped). When the counter reaches a predefined threshold, the rule will forcibly fail, thus letting the packet explore other routing opportunities.

3.3. Smooth hand-offs

To see how a nonzero slack helps the network cope with node dynamics (mobility, failures), consider the scenario shown in Fig. 8. Packets traveling between *U* and *V* are forwarded within the clouded fragment of the network. Suppose that the arrows represent neighborhoods. In a steady state, the path *A–B–C* (of length 2) is the shortest route through the cloud.

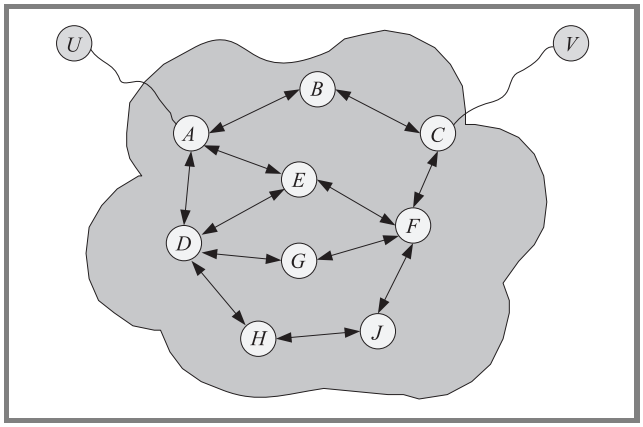

Fig. 8. A smooth handoff in TARP.

Let m be set to 1. This means that nodes *E* and *F* will also retransmit the packets because the route through them incurs a 1-hop increase over the best path. The worst thing that can happen is the disappearance of node *B*, which is a critical component of the current best path. Note, however, that this disappearance will not disrupt the traffic, because the second best path through the cloud, i.e., *A–E–F–C*, is also being used. The net outcome of this disappearance will be that a would-be duplicate arriving at *A* or *C* (from *E* or *F*), will be now *bona fide* received and forwarded towards the destination. After a short while, as the destinations update their h_b values in response to the increased number of hops along the best path, the nodes within the cloud will learn that *A–E–F–C* is the best path at the time. Then, nodes *D* and *G* will become involved as those located along the second best path (with 1-hop overhead), thus providing backup in case of subsequent mishaps.

3.4. Avoiding multiple paths with the same cost

One redundancy problem that *SPD* is unable to address is caused by possible multiple paths with the same smallest number oh hops. Consider the situation depicted in Fig. 9. Even with the most restrictive setting of the slack parameter, $m = 0$, both paths $<K_1, K_2, K_3>$ and $<L_1, L_2, L_3>$ will be occupied by the packets traveling between *S* and *D*. The duplicates will be eliminated at *A* (for the *D–S* direction) and *B* (for the direction from *S* to *D*); however, each of the K_i and L_i nodes will be consistently forwarding them because, according to *SPD*, each of those nodes is located on *a* shortest path between *S* and *D*. The problem is particularly nasty if the two rows of nodes can hear each other because then the redundant traffic contributes to the noise in their neighborhood and feeds into congestion.

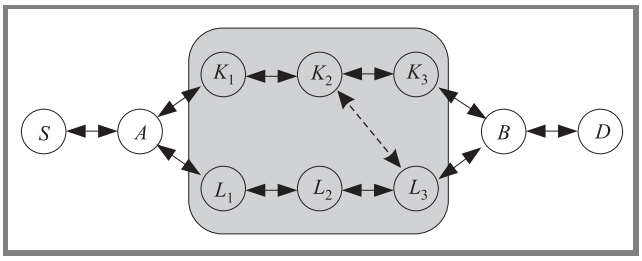

Fig. 9. Multiple paths with the same minimum cost.

To address this problem, TARP has an option whereby the packet header carries one extra bit labeled OPF (for optimal path flag). This flag is set by a forwarding node when it knows that the packet is being forwarded on one of the best paths, i.e., the SPD rule has failed non-forcibly. This means that the packet should normally reach the destination, unless some nodes have moved away or failed.

Consider nodes K_1 and L_1 in Fig. 9 receiving a packet from node *A*. Owing to the collision avoidance mechanism and randomized retransmission delays applied by the RF driver (the PHY module), one of these nodes, say K_1 will be first to re-broadcast the packet. The other node, L_1 will yield to this transmission and overhear (receive) the packet rebroadcast by K_1. Normally, that packet would be diagnosed as a duplicate and promptly discarded. However, if the OPF bit is set in the packet header, the rule in charge of discarding duplicates yields to another rule, which compares the signature of the received packet against the signatures of all packets currently queued for transmission. If a matching packet is found at L_1 and its h_f is not less than $h_f - 1$ in the received duplicate, then the packet at L_1 is dropped. In plain words, L_1 concludes that by forwarding its copy of the packet, it would not improve upon the forwarding opportunities already extended by K_1.

This mechanism will not help if the paths are disjoint, but it will kick in wherever they cross. Note that while long disjoint paths of the same length need not be rare in a realistic network, the ones for which the length is the shortest possible definitely are.

3.5. Re-casting versus point-to-point forwarding

The term "flooding" permeating the description of TARP may carry negative connotations in confrontation with the point-to-point forwarding protocols, which avoid that nasty problem by identifying precise paths within the unkempt mesh of nodes. This view is grossly misleading. First of all,

no knowledge comes out of the blue, and a point-to-point scheme is not able to deliver packets until it has discovered the paths, which operation must necessarily involve some kind of flooding. Many cases of "sales pitch" (and even some "performance studies") either ignore those stages completely or misrepresent them.

If the network is perfectly stable and static, then TARP (with zero slack) is able to achieve essentially perfect convergence to a single shortest path between any pair of peers *A* and *B* (the rare scenarios mentioned at the end of Subsection 3.4 are statistically irrelevant). This is why the term "flooding" does not adequately reflect the nature of TARP, and we prefer to call our paradigm "re-casting." On the other hand, if the network undergoes changes, then the point-to-point schemes are forced to constantly recover from lost paths, which means resorting to various forms of broadcasting and flooding. Also, the standard broadcast component of any point-to-point scheme is the persistent transmission of HELLO packets allowing all nodes to keep track of their neighborhoods.

One may argue that the point-to-point protocols are able to exploit the benefits of handshakes (like RTS-CTS-DATA-ACK of IEEE 802.11) and, in particular, circumvent the hidden/exposed terminal problem, as well as use acknowledgments on every hop, thus enhancing the reliability of communication. However, owing to the fact that most traffic in low-cost wireless networks involves packets that are very short, an RTS/CTS-type handshake is going to be completely useless and likely harmful [35]. While hop-by-hop acknowledgments can help sometimes, they are not impossible in TARP, although the problem must be considered from a slightly different angle.

In contrast to a point-to-point hop, a TARP hop has no well-defined single recipient. Notably, an internal node, i.e., one that solely forwards packets and neither generates nor absorbs them, need not even be equipped with a network address. Thus, if it cares about feedback following its forwarding action, then it would like to know whether the packet has been picked up by one or more nodes in the neighborhood, which will *bona-fide* attempt to forward it towards the destination. The approach used in our first implementation of TARP was to listen for a copy of the transmitted packet (forwarded by one of the neighbors) and interpret it as an indication of success – in addition to timers and counters used to diagnose failures. There are two problems with this solution. First, depending on the load at the forwarding node, there can be a significant delay between packet reception and retransmission. Second, to make this idea work, the destination itself has to "forward" (i.e., retransmit) all received packets, which creates unnecessary noise in its neighborhood.

A better solution employs the so-called *fuzzy acknowledgments*. When a node receives a packet, it first evaluates the rules and then, if all of them fail (i.e., the packet will be forwarded), the node responds with a short burst of RF activity (a simple unstructured packet) of a definite duration. This activity, if present, will tend to occur after a very short period of silence (analogous to SIFS in IEEE 802.11) needed by the node to evaluate the rules. When multiple recipients send their acknowledgments at (almost) the same time, the sender may not be able to recognize them as valid packets. However, it can interpret any activity (of a certain bounded duration) that follows the end of its last transmitted packet as an indication that the packet has been successfully forwarded. Even though the value of this indication may appear inferior to that of a "true" acknowledgment, it does provide the kind of feedback needed by the (informal) data-link function to assume that its responsibility for handling the packet has been fulfilled. When TARP operates with the fuzzy ACK option, any normal packet transmission is preceded by a short LBT (listen before transmit) period whose duration guarantees that fuzzy acknowledgments are not interfered with by regular packets.

Note that the implementation of fuzzy acknowledgments can be viewed as an example of inadequacy of layering in the wireless world. This is because the acknowledgment (which formally belongs to the data-link layer) can only be sent after the rules have been evaluated, i.e., the node concludes that its reception of the packet is going to contribute to its "network-layer" delivery. This is not the only place in TARP where layering gets in the way. Some rules operate best if their evaluation is postponed until the packet is about to be retransmitted, i.e., past the queuing in the data-link layer. Note that shortcuts of this kind are easily implementable within the plug-in model of VNETI.

4. Execution and emulation

The large number of generic applications for the wireless devices, combined with the obvious limitations of field testing, result in a need for emulated virtual deployments facilitating meaningful performance assessment and parameter tuning. The close relationship between PicOS and SMURPH hints at the possibility of transforming PicOS praxes into SMURPH models with the intention of executing them virtually. Until recently, one element painfully missing from the scene was a detailed model of wireless channel (SMURPH was originally intended for modeling wired networks). Once that deficiency was eliminated, the circle could be closed, i.e., SMURPH became a vehicle for executing PicOS praxes in virtual settings, practically at the source code level.

4.1. Wireless extensions to SMURPH

Owing to the proliferation of wireless channel models, and the general confusion regarding their adequacy [36], SMURPH does not implement a fixed set of channel models, but instead allows the user to easily implement flexible models, potentially capturing all the aspects of signal propagation required for a detailed functional description of diverse RF modules.

A radio channel model in SMURPH is an object of type *RFChannel.* Its role is to interconnect *transceivers*, which

provide the interface between nodes and radio channels. The built-in *RFChannel* class is intentionally open-ended: although it provides a complete functionality of sorts, that functionality is practically useless. It should be rather viewed as a generic parent type for building actual channel types, whose exact behavior is fully specified by a collection of virtual *assessment methods* provided by the user. The primary role of those methods (see Fig. 10) is to determine how a signal attenuates over distance, and how the levels of multiple signals perceived by the same recipient determine whether any of those signals can be recognized as a valid packet. In essence, they encapsulate the static (formula-like) component of the model, thus making its specification straightforward, while all the dynamic processing (like transforming the formulas into events) is hidden inside the SMURPH kernel.

```
double RFC_att (double, double, Transceiver*, Transceiver*);
double RFC_add (int, int, const SLEntry*, const SLEntry*);
Boolean RFC_act (double, double);
Boolean RFC_bot (RATE, double, double, const IHist*);
Boolean RFC_eot (RATE, double, double, const IHist*);
Long RFC_erb (RATE, double, double, double, Long);
Long RFC_erd (RATE, double, double, double, Long);
double RFC_cut (double, double);
TIME RFC_xmt (RATE, Long);
```

Fig. 10. Assessment methods of a wireless channel model in SMURPH.

For example, method `RFC_att` from Fig. 10 is responsible for calculating the received signal level, with the original signal strength and distance passed as the first two arguments. In some cases, the calculation may only depend on the distance (possibly involving randomized factors), in some others it may hinge on some intricate properties of the two transceivers involved, which are also made available to the method. Another method, `RFC_add`, carries out signal addition and is used to assess the level of interference into a reception. The multiple signals perceived by the *transceiver* are represented as an array of objects of type `SLEntry` (signal level entry). In addition to the numerical signal level, as determined by `RFC_att`, a signal level entry carries a generic user-definable attribute, which may introduce arbitrary factors into the operation, e.g., representing CDMA codes that impact the degree to which different signals contribute to the interference. Methods `RFC_bot` and `RFC_eot` are invoked to proclaim a success or failure for the action of perceiving the beginning and end of a packet. They base their decision on the so-called *interference histograms* (type `IHist`) reflecting the complete stepwise history of the interference suffered by the packet's preamble (in the first case) and the entire packet (for `RFC_eot`). `RFC_erb` and `RFC_erd` deal with bit errors and prescribe randomized occurrence of errors in preambles and packets, as well as the timing of user-definable events depending on errors. For example, the model can trigger an event on the first occurrence of a symbol error, thus aborting a packet reception in progress.

In contrast to many popular network simulators, e.g., ns-2 [30], where the fate of every packet is essentially determined at the moment of its departure, our model makes it possible for the virtual RF module to perceive a variety of events depending on dynamic levels of interference and changing predictions of bit errors. A packet reception is not a single indivisible episode, but can be split into stages affecting the module's response. Any physical action of the real counterpart of the module's virtual incarnation can be expressed and meaningfully accounted for in the model.

4.2. The virtual underlay execution engine (VUE^2)

Figure 11 shows the layout of a complete PicOS system implanted into a microcontrolled node. In particular, TARP (described in Subsection 3.1) can be seen as a plug-in to VNETI (Subsection 2.5).

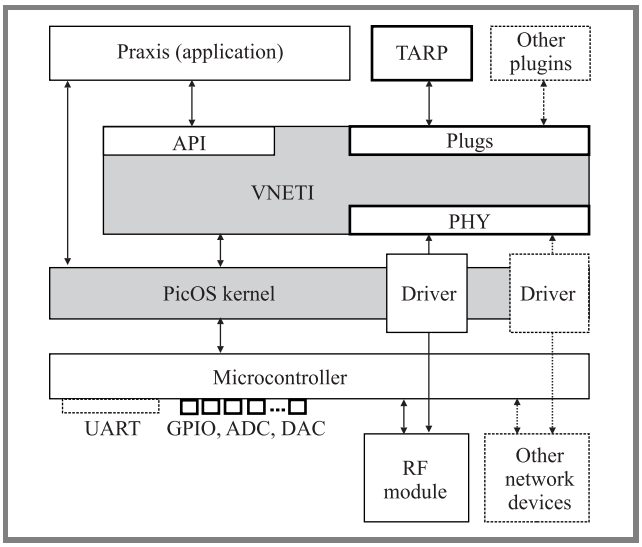

Fig. 11. System structure.

By imposing a certain software layer on SMURPH, which provides a collection of event-driven interfaces representing the environment of a PicOS praxis, and implementing a set of macros transforming PicOS keywords into their SMURPH counterparts, one can render the praxis source code acceptable as a SMURPH program. This is even possible without a formal converter[1], as long as the praxis has been coded with adherence to certain rules. This way, a PicOS praxis can be compiled and executed in the environment shown in Fig. 12, with all the physical elements of its node replaced by their detailed SMURPH models. Notably, exactly the same source code of VNETI is used in both cases.

The VUE^2 has been built with surprising ease because of the similarity in the thread models in PicOS and SMURPH. In both environments, a thread describes a finite state machine, with the state transition function specified in terms of event wait operations. The rules for aggregating such operations and waking up the threads based on the occurrence of the awaited events are practically identical in both

[1]Such a converter would be helpful, of course, and is being implemented.

systems. In SMURPH, viewed as a simulator, the awaited events are delivered by abstract objects called *activity interpreters,* while in PicOS they are triggered by actual physical phenomena (a packet reception, a character arrival from the universal asynchronous receiver/transmitter (UART), and so on).

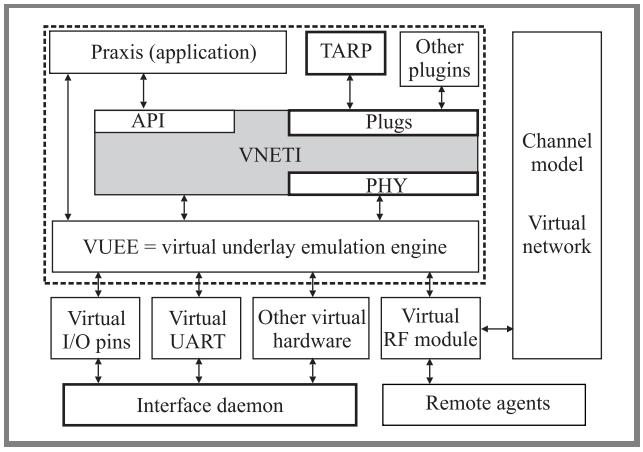

Fig. 12. The structure of a VUE^2 model.

The first significant difference between the two systems is in the interpretation of time flow. In SMURPH, time is purely virtual, which means that formally nobody cares about the actual execution time of the simulation program, but only about the proper marking of the relevant events with virtual time tags. As in all event-driven simulators, the virtual time tags have nothing to do with the real time. Consequently, the useful semantics of time for SMURPH and PicOS threads are different. The actual execution time of a SMURPH thread is essentially irrelevant (unless it renders the model execution too long to wait for the results) and all that matters is the virtual delays separating the artificially triggered events. For example, two threads in SMURPH may be semantically equivalent, even though one of them may exhibit a drastically shorter execution time than the other (due to more careful programming and/or optimization). In PicOS, however, the threads are not (just) models but they run the "real thing." Consequently, the execution time of a thread may directly influence the perceived behavior of the PicOS node. In this context, the following two assumptions make the VUE^2 project worthwhile:

1. PicOS programs are reactive, i.e., they are practically never CPU bound (see Subsection 2.6). In other words, the primary reason why a PicOS thread is making no progress is that it is waiting for a peripheral event rather than the completion of some calculation.

2. If needed (from the viewpoint of model fidelity), an extensive period of CPU activities can be modeled in SMURPH by appropriately (and explicitly) delaying certain state transitions.

In most cases, we can safely ignore the fact that the execution of a PicOS program takes time at all and only focus on reflecting the accurate behavior of the external events. With this approximation, the job of porting a PicOS praxis to its VUE^2 model can be made trivially simple. To further increase the practical value of such a model, SMURPH provides for the so-called *visualization mode* of execution. In that mode, SMURPH tries to map the virtual time of modeled events to real time, such that the user has an impression of talking to a real application. This is only possible if the network size and complexity allow the simulator to catch up with the model execution to real time. If not, a suitable slow motion factor can be employed.

A VUE^2 model can be dynamically interfaced to various *remote agents* (Fig. 12) implementing its interfaces to the real world. The behavior of those agents can be driven from scripts or manually, possibly over the Internet, by a human experimenter. For example, nodes can be powered up and down, their I/O pins can be examined and set, their sensors can be set to specific values, their UARTs or USB interfaces can be mapped to user-accessible virtual terminals or made available to other programs. In particular, an external operations support system (OSS) prepared to talk to the real network can be authoritatively tested in the virtual setup. Networking practitioners should immediately recognize the potentials of VUE^2 when applied to diverse areas of software development, from rapid prototyping to test automation. The virtual nodes can be rendered mobile in response to explicit commands or driven by programmable scenarios. The latter feature comes courtesy of SMURPH and is not VUE^2-specific.

5. Sample application blueprints

Our collection of PicOS praxes includes a set of generic wireless applications that can be easily adapted for various "typical" custom deployments. Those generic applications are called *blueprints*, even though they are fully working, demonstrable systems. For illustration, we present here two such blueprints: routing tags (*RTags*), and tags and pegs (*T&P*). They cover two large classes of applications with different mobility aspects and traffic patterns. They also illustrate how TARP, owing to its rule-driven behavior, can be optimized to different characteristics of the application.

5.1. Routing tags

Routing tags (Fig. 13) is characterized by the presence of an "elevated" node type called *master*. Any node can become master at any time, either self proclaimed or elected by other nodes. Usually, the network is partitioned among the masters with OSS interfaces for external (human or computer) operators. In a typical deployment, masters send messages to other nodes to solicit replies or to trigger some actions. This does not preclude other nodes (any nodes) exchanging messages: the traffic originating or terminating at masters is merely "highlighted," which is to say that some of TARP's parameters are optimized

for its presence. In the case of multiple OSS masters, the partitioning is functional (a given group of sensors communicates with a designated master), without being actual: the sensors and/or RTags-routers can intersperse geographically and route traffic in a group-transparent fashion.

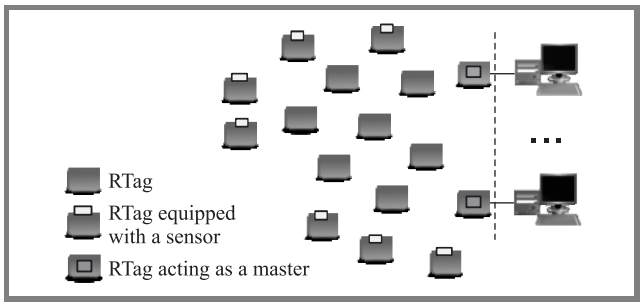

Fig. 13. Routing tags.

Masters usually send messages meant to update the context of newcomers, synchronize time-stamped functionality, and repaint fragments of the network topology for the recipients, i.e., keep the SPD caches filled with useful information (Subsection 3.2). Routing is optimized for relatively infrequent traffic and low mobility. A typical representative of this application class is an on-demand low-mobility asset monitoring system.

5.2. Tags and pegs

With tags and pegs (Fig. 14), the network consists of two types of nodes. Pegs are intentionally immobile, at least compared to tags. Some pegs can play the role of OSS gateways. Their primary purpose is to provide a kind of semi-fixed infrastructure for tracking the whereabouts of tags. Depending on the requirements, the assortment of tools facilitating this tracking may include specialized sensors deployed at tags (e.g., accelerometers, magnetic sensors) reporting their status to pegs. A significant degree of accuracy for many instances of location tracking can be achieved by measuring and correlating the received signal level (RSSI) at multiple pegs perceiving the same tag.

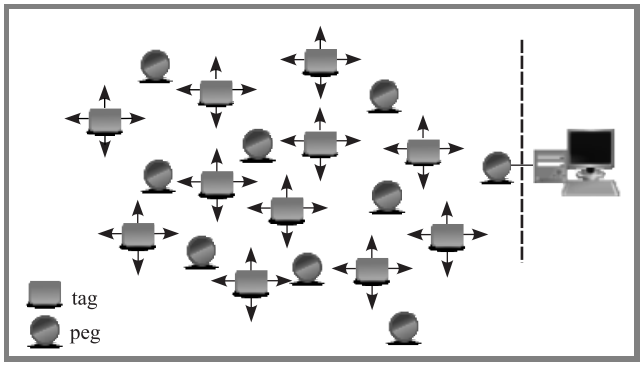

Fig. 14. Tags and pegs.

The tracking may involve various predicates applied to dynamic configurations of tags perceived in the same neighborhood. For example, a gathering of, say, 4+ people with certain attributes in an airport washroom can be detected and signaled as an event calling for special attention. The class of applications covered by *T&P* deals with mobile objects (assets, luggage, people, hazardous materials), whose mobility patterns may have to be classified by event-triggering predicates: mutual exclusion, avoidance of certain spots, time restrictions, etc.

One observation from our experiments with the various "communication modes" of the network is that practically any attempt at classification yields interesting transgressors, i.e., useful application patterns that weld fragments from seemingly distinct areas into innovative functionality. For example, *T&P* clouds embedded into an *RTags* mesh bring about the capacity for distributed self-monitoring. Envision groups of art exhibits at an exposition. A group can raise an alarm if a neighbor becomes mobile; also, it can signal the presence of an unknown member, e.g., one being removed from another area. A traveling exposition can be made self-configurable, enforcing identical setups on every stop.

We started with *RTags*, providing a generic blueprint of a monitoring system, and only after implementing *T&P* did we notice this additional and attractive distribution (or localization) of previously centralized functionality. From this point of view, our framework not only facilitates practical ad hoc networking, but also uncovers its hidden applications. Owing to the high flexibility of a TARP node, such "reconfigurations" can be often soft and dynamic, e.g., available through a sequence of commands injected into the network from an OSS agent.

6. Summary

Using the technology described in this paper, we have been able to build several practical ad hoc networks, including serious industrial deployments. By a "practical network" we understand one that works and meets the expectations of its users.

Customary, we extend the notion of practicality onto technologies, e.g., we say that Ethernet technology is practical, even if some of its botchy specimens fail. Networks acquire practicality via technological progress and industrial acceptance, but this acquisition need not be universal. Ethernet or IP networks are indisputably practical, so are ATM and Bluetooth, even if their cases illustrate the fact that practicality not always follows common sense. On the other hand, IP extensions (meant to make it a "one for all" choice), should, after all these years, be denied practicality. So, we are afraid, must some wireless ad hoc networking schemes, notably ZigBee®, despite powerful industrial sponsors behind them. In the latter case, most of the harm is inflicted by confusing a general scheme with a complete

solution, of which the scheme is merely a (likely quite suboptimal) part.

We have presented here a technology that, in our opinion, makes ad hoc networks practical, i.e., functional and deployable in a variety of industrial frameworks. While we do not claim that ours is the only possible approach to practical ad hoc networking, we couldn't find a better one despite honest attempts. Our present library of application blueprints (working, demonstrable, open-ended data-exchange patterns) makes us confident that the combination of tools and methodologies comprising our platform is powerful enough to handle many practically interesting cases of distributed sensing, monitoring, industrial process control, and so on. We are proud of the fact that every detail described in this paper has found its way into real (deployed) ad hoc networks. Large fragments of our research and development were stimulated, or even directly triggered, by the findings and wishes of their users.

References

[1] K. Altisen, F. Maraninchi, and D. Stauch, "Aspect-oriented programming for reactive systems: a proposal in the synchronous framework", Res. Rep. no. tr-2005-18, Verimag CNRS, Nov. 2005.

[2] F. Balarin *et al.*, "Scheduling for embedded real-time systems", *IEEE Des. Test of Comput.*, vol. 15, no. 1, pp. 71–82, 1998.

[3] M. Berard, P. Gburzyński, and P. Rudnicki, "Developing MAC protocols with global observers", in *Proc. Comput. Netw.'91*, Kraków, Poland, 1991, pp. 261–270.

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

[5] D. R. Boggs, J. C. Mogul, and C. A. Kent, "Measured capacity of an Ethernet: myths and reality", WRL Res. Rep. 88/4, Digital Equipment Corporation, Western Research Laboratory, 100 Hamilton Avenua, Palo Alto, California, 1988.

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

[7] O. J. Dahl and K. Nygaard, *Simula: A Language for Programming and Description of Discrete Event Systems. Introduction and User's Manual*, 5th ed. Oslo: Norwegian Computing Center, 1967.

[8] W. Dobosiewicz and P. Gburzyński, "Improving fairness in CSMA/CD networks", in *Proc. IEEE SICON'89*, Singapore, 1989.

[9] W. Dobosiewicz and P. Gburzyński, "Performance of piggyback Ethernet", in *Proc. IEEE IPCCC*, Scottsdale, USA, 1990, pp. 516–522.

[10] W. Dobosiewicz and P. Gburzyński, "On the apparent unfairness of a capacity-1 protocol for very fast local area networks", in *Proc. Third IEE Conf. Telecommun.*, Edinburgh, Scotland, 1991.

[11] W. Dobosiewicz and P. Gburzyński, "An alternative to FDDI: DPMA and the pretzel ring", *IEEE Trans. Commun.*, vol. 42, pp. 1076–1083, 1994.

[12] W. Dobosiewicz and P. Gburzyński, "On two modified Ethernets", *Comput. Netw. ISDN Syst.*, pp. 1545–1564, 1995.

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

[14] W. Dobosiewicz and P. Gburzyński, "The spiral ring", *Comput. Commun.*, vol. 20, no. 6, pp. 449–461, 1997.

[15] W. Dobosiewicz, P. Gburzyński, and V. Maciejewski, "A classification of fairness measures for local and metropolitan area networks", *Comput. Commun.*, vol. 15, pp. 295–304, 1992.

[16] W. Dobosiewicz, P. Gburzyński, and P. Rudnicki, "On two collision protocols for high-speed bus LANs", *Comput. Netw. ISDN Syst.*, vol. 25, no. 11, pp. 1205–1225, 1993.

[17] D. Drusinsky, M. Shing, and K. Demir, "Creation and validation of embedded assertions statecharts", in *Proc. 17th IEEE Int. Worksh. Rap. Syst. Protot.*, Chania, Greece, 2006, pp. 17–23.

[18] D. Dubhashi *et al.*, "Blue pleiades, a new solution for device discovery and scatternet formation in multi-hop Bluetooth networks", *Wirel. Netw.*, vol. 13, no. 1, pp. 107–125, 2007.

[19] P. Gburzyński, *Protocol Design for Local and Metropolitan Area Networks*. Upper Saddle River: Prentice-Hall, 1996.

[20] P. Gburzyński and J. Maitan, "Simulation and control of reactive systems", in *Proc. Wint. Simul. Conf. WSC'97*, Atlanta, USA, 1997, pp. 413–420.

[21] P. Gburzyński, J. Maitan, and L. Hillyer, "Virtual prototyping of reactive systems in SIDE", in *Proc. 5th Eur. Concur. Eng. Conf. ECEC'98*, Erlangen-Nuremberg, Germany, 1998, pp. 75–79.

[22] M. Gunes, U. Sorges, and I. Bouazizi, "ARA – the ant-colony based routing algorithm for manets", in *Proc. Int. Worksh. on Ad Hoc Netw. IWAHN*, Vancouver, Canada, 2002.

[23] D. Harel, "On visiual formalisms", *Commun. ACM*, vol. 31, no. 5, pp. 514–530, 1988.

[24] J. Hui, "TinyOS network programming (version 1.0)", TinyOS 1.1.8 Documentation, 2004.

[25] D. B. Johnson and D. A. Maltz, "Dynamic source routing in ad hoc wireless networks", in *Mobile Computing*, T. Imielinski and H. Korth, Eds. Norwell: Kluwer, 1996, vol. 353.

[26] P. Levis *et al.*, "TinyOS: an operating system for sensor networks", in *Ambient Intelligence*, W. Weber, J. M. Rabaey, and E. Aarts, Eds. Springer, 2005, pp. 115–148.

[27] J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris, "A scalable location service for geographic ad hoc routing", in *Proc. ACM/IEEE Int. Conf. Mob. Comput. Netw. MOBICOM'00*, Boston, USA, 2000, pp. 120–130.

[28] X. Ling *et al.*, "Performance analysis of IEEE 802.11 DCF with heterogeneous traffic", in *Proc. Consum. Commun. Netw. Conf.*, Las Vegas, USA, 2007, pp. 49–53.

[29] J. Liu and E. A. Lee, "Timed multitasking for real-time embedded software", *IEEE Contr. Syst. Mag.*, vol. 23, no. 1, pp. 65–75, 2003.

[30] D. Mahrenholz and S. Ivanov, "Real-time network emulation with ns-2", in *Proc. Eighth IEEE Int. Symp. Distrib. Simul. Real-Time Appl.*, Budapest, Hungary, 2004, pp. 29–36.

[31] V. D. Park and M. S. Cors, "A performance comparison of TORA and ideal link state routing", in *Proc. IEEE Symp. Comput. Commun.'98*, Athens, Greece, 1998.

[32] C. Perkins, E. Belding Royer, and S. Das, "Ad-hoc on-demand distance vector routing (AODV)", Febr. 2003, Internet Draft: draft-ietf-manet-aodv-13.txt.

[33] C. E. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance vector routing (DSDV) for mobile computers", in *Proc. SIGCOMM'94*, London, UK, 1993, pp. 234–244.

[34] C. E. Perkins and E. M. Royer, "Ad-hoc on-demand distance vector routing (AODV)", in *Proc. IEEE Worksh. Mob. Comp. Syst. Appl. WMCSA*, New Orleans, USA, 1999, pp. 90–100.

[35] A. Rahman and P. Gburzynski, "Hidden problems with the hidden node problem", in *Proc. 23rd Bienn. Symp. Commun.*, Kingston, Canada, 2006, pp. 270–273.

[36] T. K. Sarkar, Z. Ji, K. Kim, A. Medouri, and M. Salazar-Palma, "A survey of various propagation models for mobile communication", *IEEE Anten. Propag. Mag.*, vol. 45, no. 3, pp. 51–82, 2003.

[37] K. Schwan and H. Zhou, "Dynamic scheduling of hard real-time tasks and real-time threads", *IEEE Trans. Softw. Eng.*, vol. 18, no. 8, pp. 736–748, 1992.

[38] C.-K. Toh, "A novel distributed routing protocol to support ad-hoc mobile computing", in *Proc. IEEE 15th Ann. Int. Phoenix Conf. Comp. Commun.*, Phoenix, USA, 1996, pp. 480–486.

[39] B. Yartsev, G. Korneev, A. Shalyto, and V. Ktov, "Automata-based programming of the reactive multi-agent control systems", in *Int. Conf. Integr. Knowl. Intens. Multi-Agent Syst.*, Waltham, USA, 2005, pp. 449–453.

Paweł Gburzyński holds a Ph.D. in informatics from the University of Warsaw, Poland. Before emmigrating to Canada in 1984, he had been involved in a number of software development projects in Poland, including Loglan, in collaboration with Dr. Andrzej Salwicki and his team. Since 1985 he has been with the faculty of the Department of Computing Science, University of Alberta, Canada, where he is a Professor. In 2002, he cofounded Olsonet Communications Corporation, where he acts as Chief Scientist. Dr. Gburzynski's contributions include SMURPH (a high-fidelity modeling/emulation package for communication systems) and PicOS (an operating system for embedded platforms). He has devised a number of communication protocols, including ad hoc wireless routing schemes and custom protocols for high-performance systems, and implemented many practical wireless networking solutions. His research interests are in telecommunication, embedded systems, operating systems, simulation and performance evaluation. As a hobby, he develops effective anti-spamming tools.

e-mail: pawel@cs.ualberta.ca
Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8 Canada

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

e-mail: wlodek@olsonet.com
Olsonet Communications Corporation
51 Wycliffe Street
Ottawa, Ontario, K2G 5L9 Canada