



Ad-hoc networking with low-cost devices: how to do it right

Pawel Gburzynski¹ and Wlodek Olesinski²

¹ University of Alberta, Department of Computing Science,
Edmonton, Alberta, CANADA T6G 2E8

² Olsonet Communications Corporation,
51 Wycliffe Street, Ottawa, Ontario, CANADA K2G 5L9

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. Commercially available nodes for ad-hoc networking (somewhat inappropriately called “motes”)³ are in fact quite serious computers with tens of 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.

1 Introduction

The vast number of academic contributions to the area of ad-hoc wireless networking has left a surprisingly tiny footprint in the practical world. For once, the industry is not much smarter. The problem is that 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 an intelligent ad-hoc wireless system.

The plethora of popular ad-hoc routing schemes proposed and analyzed in the literature [14, 1, 13, 10, 8, 12, 16, 6], address 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

³ See for example <http://www.xbow.com/>.

than “recommended” amount of RAM can still fulfill its obligation to the network, but, perhaps, at a reduced (albeit manageable and acceptable) quality. 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: 1) a large footprint and, consequently, non-trivial cost; 2) 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 [4], it is commonly agreed that the role of this technology is reduced to creating small personal area hubs. ZigBee® (based on AODV [15]) 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 reusability, 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 [9], 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 requirements enable new applications and trigger even more demand.

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 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 internetwork them in a truly ad-hoc fashion.

2 The operating system

The most serious problem with implementing multitasking within tiny RAM is the stack space, which must be rigidly pre-allocated to every task. Notably, this space is wasted from the viewpoint of the application: it is merely a “working area” needed to build the high-level structure of the program. One radical solution, e.g., adopted in TinyOS [7, 9], is simply to eliminate multitasking. Essentially, TinyOS defines two types of activities: event handlers and *tasks*, which are simply chunks of code that cannot be preempted by (and thus cannot coexist with) other tasks.

Our operating system, dubbed PicOS, does implement multitasking, whereby tasks explicitly declare preemption points in the form of *states*. A PicOS task looks like a finite state machine (FSM) that changes its states in response to events. The CPU is multiplexed among the multiple concurrent tasks, but only at state boundaries.

```

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

```

Fig. 1. A sample task (strand) code in PicOS.

For illustration, consider the sample task code shown in Figure 1. Its different states are marked by the `entry` statements. Whenever a task is assigned the CPU, its code function is called in the *current state*, i.e., at one specific entry point.

Task blocking always involves an explicit state indicating where the task should be resumed when the blocking condition disappears. For example, function `tcv_rnp` in state `RC_TRY` attempts to acquire an input packet for processing. If no packet is available, the function does not return: the process loses the CPU and will be resumed in state `RC_TRY` when a packet becomes available. The sequence in state `RC_PASS` (`when`, `delay`, `release`) exemplifies a more explicit state transition scenario. With `when`, the task 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 a timer to go off after 1000

milliseconds, which will resume the task in state `RC_LOCKED`. Finally, `release` explicitly gives up the CPU: the task becomes blocked until any of the awaited conditions occurs.

The above paradigm of organizing tasks in PicOS has proven very versatile, expressive, structural, and self-documenting. The inherent FSM layout of the praxis comes for free and can be mechanically transformed, e.g., into a state-chart [3], for easy comprehension or verification. Owing to the fact that a blocked task needs no stack space, all tasks 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.

So far, PicOS has been implemented on the MSP430 microcontroller family and on eCOG1 from Cyan Technology. The size of the task control block (PCB) needed to describe a single PicOS task is adjustable by a configuration parameter, depending on the number of events E that a single task may want to await simultaneously. The standard setting of this number is 3, which is sufficient for all our present applications and protocols. The PCB size in bytes is equal to $8 + 4E$, which yields 20 bytes for $E = 3$. The number of tasks 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 tasks (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.

3 Communication

Most routing protocols for ad-hoc wireless networks 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 [14, 1] or reactive [13, 10, 8, 12, 16, 6], 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 multicast, 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 [15], 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 from previous path discoveries, a node receiving such a request may decide to forward it elsewhere, or, possibly, respond backward 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, this generic approach requires the nodes to store a potentially sizable amount of elaborate routing information, which, typically, 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.

The idea behind our solution, dubbed TARP (for Tiny Ad-hoc Routing Protocol), is to embrace fuzziness as a useful feature and to 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 [11]. TARP, in contrast, turns it into an advantage.

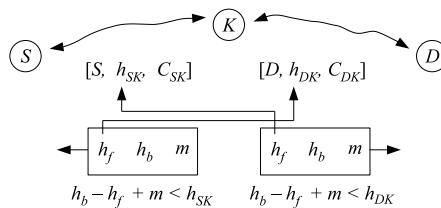


Fig. 2. The rule for selective packet discard (SPD).

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 traveled by a packet. The key to the success of our variant of flooding is the rule that brings the paths traveled by forwarded packets down to a narrow (but intentionally fuzzy) stripe of nodes along the shortest route.

Consider the three nodes shown in Figure 2. 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 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. We assume that a node that does not have enough information to make an educated decision, forwards the packet (unless the obvious rules say otherwise).

Note that this mode of operation is not fundamentally less economical than one based on point-to-point forwarding (in a traditional ad-hoc routing scheme). This is because before a point-to-point system can commence forwarding, it has to learn the routes one way or the other. In a traditional scheme, there are distinct phases of knowledge acquisition and recovery from its decay, which are separate from the actual forwarding. In our approach, the acquisition and maintenance of the requisite knowledge happens together with forwarding (as its byproduct) and acts constantly to preserve and improve its quality.

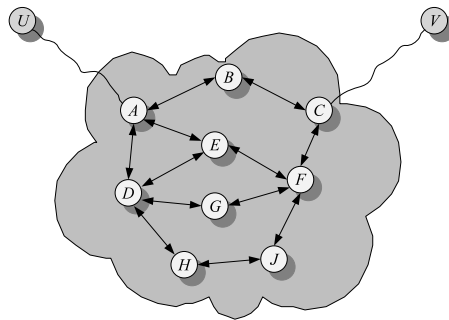


Fig. 3. A smooth handoff in TARP.

Owing to the inherent imperfections of the ad-hoc wireless environment, K should not be too jumpy with negative decisions. TARP includes a slack parameter m in the inequality (as shown in Figure 2). When $m > 0$, the rule will allow

the node to forward the packet when the path passing through it appears to be slightly longer (by up to m hops) than the currently believed shortest path. This way, m introduces controllable redundancy into the scheme. To see how it helps the network cope with node dynamics (mobility, failures), consider the scenario shown in Figure 3. 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. 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 moment.

To cope with more drastic changes in the network configuration, every once in a while, a routing node forwards the packet to be otherwise dropped (based on the inequality in Figure 2). This happens with frequency inversely proportional to the margin by which the inequality holds.

An important property of the rule-based routing system of TARP is that the way all rules are implemented makes them automatically scalable to the amount of RAM available at the node. This is because their behavior is driven by caches, and the lack of information in the cache forces the rule to decide that the packet *should not* be dropped on its account. Thus, a node with too little memory may make suboptimal decisions, i.e., forwarding more packets than it absolutely should, but it will never break the connectivity.

4 Praxis structure

The interface of a PicOS application (praxis) to the outside world is governed by a powerful module called VNETI (for Virtual NETwork Interface). VNETI offers to the praxis a simple collection of 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. For example, TARP is just a plug-in to VNETI. The actual connection to the physical transceiver module is encapsulated and abstracted into a PHY—an easily exchangeable module with a minimalistic program interface. Multiple plug-ins and physical interfaces can coexist within the same system configuration.

The interaction of all system components is shown in Figure 4. VNETI provides built-in means for buffering packets, moving them among different queues, and locating the relevant components of their payloads. A plug-in may intercept

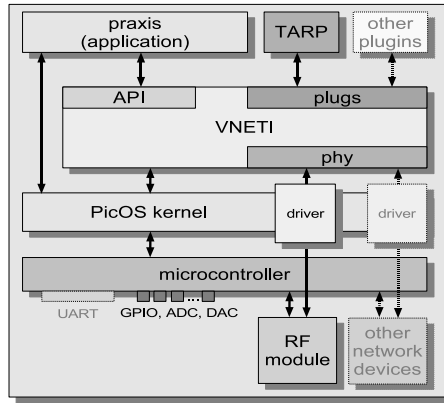


Fig. 4. System structure.

some or all packets arriving from the network or from the application, modify them, insert new packets into queues, and so on. All this happens in a way that preserves the built-in API and PHY interfaces, which are formally independent of the configuration of plug-ins. In contrast to the traditional layered approach, a VNETI plug-in acts in a “holistic” manner, which means that it can see and affect the *whole* of the packet’s structure and fate. The key to power, convenience, simplicity, and safety (resilience to errors) of this approach lies in the careful design of the available tools.

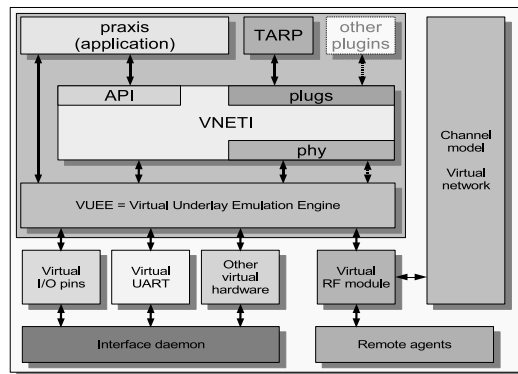


Fig. 5. The structure of a VUE² model.

An important stage of the application design process is its testing, which is greatly facilitated by VUE² (Virtual Underlay Emulation Engine) based on SMURPH [5]. VUE² makes it possible to execute the complete application in an artificial environment, where multiple nodes communicate over virtual wire-

less channels and, possibly, talk to real-life OSS agents. The latter means that the complete functionality of the target networked application can be verified virtually, including its collaboration with real external programs.

The structure of a VUE² model is depicted in Figure 5. Owing to the fact that PicOS's programming paradigm directly descends from SMURPH [2], it is reasonably easy for VUE² to understand the structure and behavior of a PicOS program and simulate its execution. In particular, exactly the same VNETI (the same source code) executes under PicOS and VUE². That source code, together with the praxis program, are mechanically transformed into a SMURPH program, then compiled and linked to the libraries providing the models of standard components, including wireless channels and peripheral devices, like UARTs, A-D converters, etc. If the network is not excessively large, the model is able to execute in real time, meaning that the observed time flow coincides with its flow in real life. A slow-motion factor can be applied to models of large networks.

5 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.

References

1. T-W. Chen and M. Gerla. Global state routing: a new routing scheme for ad-hoc wireless networks. In *Proceedings of ICC'98*, June 1998.
2. W. Dobosiewicz and P. Gburzynski. From simulation to execution: on a certain programming paradigm for reactive systems. In *Proceedings of the First International Multiconference on Computer Science and Information Technology (FIMCSIT'06)*, pages 561–568, Wisla, Poland, nov 2006.
3. D. Drusinsky, M. Shing, and K. Demir. Creation and validation of embedded assertions statecharts. In *Proceedings of 17th IEEE International Workshop Rapid Systems Prototyping*, pages 17–23. IEEE CS Press, 2006.
4. D. Dubhashi et al. Blue pleiades, a new solution for device discovery and scatternet formation in multi-hop Bluetooth networks. *Wireless Networks*, 13(1):107–125, 2007.
5. P. Gburzynski and I. Nikolaidis. Wireless network simulation extensions in SMURPH/SIDE. In *Proceedings of the 2006 Winter Simulation Conference (WSC'06)*, Monterey, California, dec 2006.
6. M. Gunes, U. Sorges, and I. Bouazizi. ARA—the ant-colony based routing algorithm for manets. In *Proceedings of International Workshop on Ad-hoc Networking (IWAHN)*, Vancouver, British Columbia, Canada, August 2002.
7. J. Hui. Tinyos network programming (version 1.0), 2004. TinyOS 1.1.8 Documentation.
8. D. B. Johnson and D. A. Maltz. Dynamic Source Routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.
9. P. Levis et al. TinyOS: An operating system for sensor networks. In W. Weber, J.M. Rabaey, , and E. Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer, 2005.
10. J. Li, J. Jannotti, D. De Couto, D. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM' 00)*, pages 120–130, 2000.
11. X. Ling et al. Performance analysis of IEEE 802.11 DCF with heterogeneous traffic. In *Proceedings of Consumer Communications and Networking Conference*, pages 49–53, Las Vegas, NV, January 2007.
12. V.D. Park and M.S. Cors, on. A performance comparison of TORA and ideal link state routing. In *Proceedings of IEEE Symposium on Computers and Communications '98*, June 1998.
13. C. Perkins, E. Belding Royer, and S. Das. Ad-hoc On-demand Distance Vector Routing (AODV), February 2003. Internet Draft: draft-ietf-manet-aodv-13.txt.
14. C. E. Perkins and P. Bhagwat. Highly dynamic Destination-Sequenced Distance Vector routing (DSDV) for mobile computers. In *Proceedings of SIGCOMM'94*, pages 234–244, August 1993.
15. C. E. Perkins and E. M. Royer. Ad-hoc On-demand Distance Vector Routing (AODV). In *Proceedings of the IEEE workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 90–100, 1999.
16. C-K Toh. A novel distributed routing protocol to support ad-hoc mobile computing. In *Proceedings of IEEE 15th Annual International Phoenix Conf. on Comp. and Comm.*, pages 480–486, March 1996.