

# TinyNET: A Tiny Network framework for TinyOS

Angelo P. Castellani, Paolo Casari and Michele Zorzi

Department of Information Engineering, University of Padova

Via G. Gradenigo, 6/B, 35131 Padova, Italy

Email: {castellani, casarip, zorzi}@dei.unipd.it

We present TinyNET, a modular framework that allows development and quick integration of network protocols and applications for Wireless Sensor Networks (WSNs) in TinyOS. The main advantages of using our modular framework can be summarized as follows: *i*) makes it easier to build applications by allowing easy reuse of software modules; *ii*) provides any protocol and application with a layered network interface that encompasses the whole stack, while still allowing cross-layer operations and exchange of parameters; *iii*) allows fast reconfiguration of applications through new protocols and functionalities, that transparently become a part of the layered network stack. Our framework operates on top of TinyOS, but below the user application modules, and is completely transparent (in the sense that TinyOS module binding directives are intercepted and used to place the module within the framework).

The communication abstraction employed in TinyOS is the Active Message (AM) model. The AM paradigm straightforwardly allows to share the radio interface by binding applications to a single AM type of the Active Message subsystem. Applications employ available interfaces to control the radio subsystem, e.g., to power it on/off and, by platform specific commands, read link quality indicators (TX power, RSSI, LQI). Directly putting an application in control of the radio subsystem is a valid approach only if the application itself is very simple; in case a more complex system should be built, a top-down approach is preferred, which requires to design the architecture and modules of the system before developing the system itself. However, network applications are usually designed as a holistic module which is tightly integrated with TinyOS, with hardly reusable parts and usually incorporating platform-specific code. This is also due to the structure of TinyOS, whose development architecture does not encourage structured modular design. Furthermore, there is no logical network architecture available for TinyOS, which represents a drawback to open contributions of network protocols and applications. TinyNET is a network framework designed to help fill the gap between TinyOS itself and any kind of networked system built on top of it.

TinyNET exploits nesC to split any networked system into two parts: the application layer and the network layer. The *application layer* is similar to TinyOS's standard developing entry point, with the additional feature that every application module represents a single, independent process in the network system. Utility interfaces have been built to perform such operations as radio state control and channel selection; in addition, the control of the radio subsystem has been centralized, thus providing the ability to intercept any control request. In turn, this common entry point facilitates the development of control concurrency resolution techniques, e.g., through independent locks to be imposed on the radio by those applications that require exclusive access to this resource. The *network layer* is

instead a novel layer which contains any modules that require full access to every packet received and/or transmitted by the node. The network layer is a direct development entry point for new network protocols to be inserted in the framework, transparently to applications. This layer also support ordered access of protocols to transmitted/received packets, and provides full control over the packet itself, e.g., any module can change the field structure of the packets if required. To strengthen the differentiation among the application and network layer, they are programmed inside separate files, allowing system integrators to easily combine application and network components.

As to ROM occupancy, the use of TinyNET increased the BlinkToRadio size by 3.5kB, reaching a total size of about 15kB. However, it should be noted that this overhead is fixed, and depends neither on which nor on how many applications are loaded, nor on how many network modules have been wired to the framework. The RAM occupancy overhead depends on the buffer size of the scheduling queues as set up in configuration files, plus about 60B of static variables allocated by the framework itself.

After testing TinyNET's memory footprint, we still needed to experience practical advantages brought by using TinyNET in comparison to the standard TinyOS programming approach. To this end, we have built a more complex system, featuring several applications, networking and communication modules. A multi-hop environmental monitoring and querying system using 6LowPAN has been chosen in this regard, as it is complex enough to prove the advantages of using the TinyNET framework. We highlight that the focus of the work was to prove that TinyNET allows easy and straightforward implementation of these modules, compared to TinyOS, rather than on collecting performance metrics related to the system itself.

Our experience is that TinyNET represents a change of perspective with respect to the usual development of TinyOS applications, as it endows TinyOS developers with the chance to recover the well known divide-and-conquer design strategy for complex systems. In other words, TinyNET allows to subdivide the development into many simpler independent parts, each to be handled separately. Using the provided framework interfaces, the development of components was fast, allowing straightforward implementation, and easy debug; in particular, debug is facilitated by concentrating on a single module instead of inspecting a monolithic code. Maintaining the software is also substantially easier, as each component can be replaced, integrated with other components, or deleted. Thanks to the modularity of the developed system, every component can be swapped with no further adaptations; additionally, new features can be added on top of the already available software by creating new, separate modules, and performing the proper wiring.