Drop the Phone and Talk to the Physical World: Programming the Internet of Things with Erlang

Alessandro Sivieri, Luca Mottola, Gianpaolo Cugola
Politecnico di Milano
DeepSE Group, Dipartimento di Elettronica e Informazione
Piazza L. da Vinci, 32 Milano, Italy
{*sivieri,mottola,cugola*}*@elet.polimi.it*

***Abstract*—We present ELIoT, an Erlang-based development framework expressly conceived for heterogeneous and massively decentralized sensing/actuation systems: a vision commonly regarded as the "Internet of Things". We choose Erlang due to the functional high-level programming model and the native support for concurrency and distributed programming. Both are assets when developing applications as well as system-level functionality in our target domain. Our design enriches the Erlang framework with a custom library for programming sensing/actuation distributed systems along with a dedicated run-time support, while we wipe off unnecessary language and run-time features. We couple the resulting platform with ad-hoc tools for simulation and testing, supporting developers throughout the development cycle. We assess our solution by implementing three sensor network distributed protocols. A comparison with traditional sensor network programming platforms demonstrates the advantages in terms of terseness of code, readability, and maintainability.**

***Keywords*-distributed systems, Internet of Things, programming languages, frameworks**

I. Introduction

The vision of an "Internet of Things" (IoT)—an ensemble of heterogeneous devices embedded within the environment—brings along a plethora of software engineering challenges [1]. Wireless sensor networks, by many regarded as a forerunner of the future IoT and a key component thereof, already exemplified some of these issues [2], [3]. The increased complexity of IoT applications, along with the sheer number of heterogeneous resource-constrained devices involved, is going to worsen such state of affairs.

Motivation. To date, embedded sensing/actuation systems are mostly developed using low-level languages akin to C, atop the operating system facilities [2]. This provides full control of the scarce resources available and thus opens up significant opportunities for optimizations. However, it also typically leads to implementations that are difficult to test, to maintain, and to port to different platforms [4].

The literature includes several attempts to raise the level of abstraction, some to the point of considering the network of embedded sensor and actuators as a single computational unit—an approach commonly termed as "macroprogramming" [2]. Such solutions, however, often trade generality and performance for ease of programming. Finding a middle ground proves to be extremely difficult [5].

Erlang. We maintain that the Erlang language provides an ideal stepping stone to address the issues above:

- it combines a *functional core* with dynamic typing and pattern-matching to guide the computation and to access data, supporting a form of concise declarative programming that already proved effective in sensor network development [6];
- it adopts an *actor-like concurrency model* [7]: different processes communicate only though message passing independently of their physical location, effectively *masking distribution* to a great extent and thus facilitating developing massively decentralized functionality;
- owing to its origins, it provides features expressly designed for *embedded systems programming*, e.g., the ability to pattern-match on bit streams, enabling high-level descriptions of packet manipulation functions;
- in most existing implementations, Erlang is an *interpreted* language: this naturally addresses portability issues and allows code to be easily hot-swapped, supporting long running applications characterized by transient interactions that are redefined on the fly.

Section II provides a brief Erlang primer to exemplify some of the features above.

ELIoT. Nevertheless, Erlang is by no means directly applicable to developing IoT sensing/actuation systems. For example, the semantics of Erlang's inter-process communication leverages reliable point-to-point communication. This does not capture the many-to-many localized interactions of IoT scenarios, besides being typically provided using full-fledged TCP/IP stacks rarely found in resource-constrained IoT devices. The language evolution and inclusion of increasingly sophisticated libraries also led existing run-time support systems to grow accordingly, ultimately posing significant requirements on the underlying hardware platform.

We address these issues with ELIoT[1], our Erlang-based development framework for IoT systems, described in Sec-

[1] ELIoT is ErLang for the Internet of Things.

```
1  start(X) ->
2      Pid = spawn(fun power2/0),
3      Pid ! {self(), X},
4      receive
5          {Sender, Result} ->
6              Result
7      end.
8
9  power2() ->
10     receive
11         {Sender, Number} ->
12             Sender ! {self(), Number * Number}
13     end.
```

Figure 1. Sample Erlang code to compute the power of two.

tion III. We provide dedicated libraries for localized many-to-many inter-process communication that developers use to describe both system-level and application functionality. By removing unnecessary features, we also significantly reduce the hardware requirements of the run-time system: our current implementation runs on embedded devices with a few megabytes of RAM, the size of a gum stick, and commercially available for less than 100$. The resulting platform is thus immediately applicable in scenarios such as smart energy [8] and remote patient monitoring [9].

To support developers during the validation phase, we complement the framework with a dedicated simulator running un-modified ELIoT code. Leveraging Erlang's concurrency model, our simulator allows developers to start validating the system in a fully simulated environment, and then to progressively and transparently migrate to real hardware by running mixed deployments including both simulated and real nodes. This way, developers retain visibility into the system state through the simulator, and yet are able to check the execution of real hardware, e.g., w.r.t. timings and unreliability of the wireless channel.

Section IV reports on our preliminary assessment of ELIoT's effectiveness. We consider three sensor network distributed protocols as representative of the complexity involved in embedded sensing/actuation systems. We implement and test them with ELIoT. A comparison against functionally-equivalent versions of the same protocols implemented atop TinyOS [10] and Contiki [11]—the defacto standard programming platforms for sensor networks—reveals advantages in both readability and maintainability of the implementations.

Following a brief survey of related work in Section V, we conclude the paper in Section VI by illustrating the steps ahead in our research agenda.

II. Erlang in a Nutshell

We exemplify some of Erlang's features to provide a more concrete illustration to readers unfamiliar with the language.

Figure 1 illustrates a simple snippet of Erlang code to compute the power of two. The base number (**X**) is given as input to the **start** function (line 1). The function then spawns a second process (line 2) that starts executing the **power2** function. Erlang processes are lightweight computational units handled directly by the interpreter and are not mapped directly onto system processes or threads. This way, process creation is faster and consumes less resources.

According to the actor model, Erlang processes cannot share memory. The only way to exchange data is via asynchronous message-passing. Each process has an associated persistent mailbox, where messages are read in FIFO order. An example is shown in line 3, where the process executing **start** sends a message carrying its own identifier and the input number to the process previously spawned, using the **!** sign. At the other end, the **receive** keyword (line 10) lists a set of patterns to process incoming messages, executed according to *pattern matching*. In this example, the receiving process simply bounces back another message with its own identifier and the result of the computation (lines 11-12). Upon receiving such message, the first process exits by returning the final result (lines 4-7). Such model of computation inherently fosters the development of loosely coupled functionality.

Beyond this example, Erlang features several characteristics of functional languages that favor reliable implementations of distributed software. For instance, it features a single-assignment semantics, whereby variables can be assigned only once and maintain the same value throughout the execution, improving readability. Moreover, function evaluation is side-effect free, enabling concurrent execution. Higher-order functions (functions receiving or returning other functions as parameters) are permitted, allowing the development of generic functions that specialize at run-time based on the actual parameters. Finally, Erlang supports list comprehension, which makes it possible to manipulate lists directly by applying predicates or functions.

The OTP (Open Telecom Platform) library also adds several functionality to the core language. It introduces the concept of *supervisor processes*, computational units external to the main application that monitor its execution and apply counter-measures should faults be detected. The library also provides a configuration facility to ease the hot-swapping of existing functionality and process design patterns implementing generic behaviors that developers customize to their application-specific needs.

At run-time, the Erlang interpreter hides a great part of the complexity due to distribution. As example, messages are delivered across different processes independently of their physical location: it is completely transparent to developers if a message originates from a process in the same or different host. This allows developing and testing applications first in a local setting, and then to progressively move to a distributed setting with (almost) no changes to the code. We leverage this feature to ease the testing of IoT embedded sensing software, as described next.

```
1  -module(example_module).
2  -export([start_link/0, example_function/0]).
3  -define(SLEEP, 60000).
4  -define(NODE, 1).
5  -define(TEMPERATURE, 1).
6
7  start_link() ->
8    Pid = spawn_link(?MODULE, example_function, []),
9    register(example_proc, Pid),
10   eliot_api:export(Pid),
11   {ok, Pid}.
12
13 example_function() ->
14   timer:send_after(?SLEEP, read),
15   receive
16     read ->
17       Value = eliot_sensors:read(?TEMPERATURE),
18       eliot_api:bcast_send(example_proc,
19                            {?NODE, Value});
20     {SourceId, RSSI, Pload} when ?NODE==1 ->
21       eliot_api:send(example_proc,
22                      {node@192.168.1.1,Pload});
23     {SourceId, RSSI, Pload} ->
24       eliot_api:bcast_send(example_proc,
25                            {?NODE, Pload})
26   end,
27   example_function().
```

Figure 2. ELIoT program implementing a sense-and-send pattern.

III. ELIoT

ELIoT includes three core parts: *i)* a small library to develop decentralized sensing/actuation systems; *ii)* a custom interpreter tailored to resource-poor IoT devices; and *iii)* a dedicated simulator to test the implementations in a fully or partially simulated environment.

A. Library

Figure 2 illustrates the use of the ELIoT library to develop a simple sense-and-send data collection functionality, in fact a pattern seen in many sensor network implementations [2]. The code, with the proper filtering of duplicate packets—not shown in the picture, is already sufficient to implement such behavior across multiple hops.

Following the specification of module name, exported processes, and constants, the entry point of the module (line 7) spawns a new local process to execute the main loop (line 8), registers the spawned process locally (line 9), and makes it reachable from the network using the **`export`** function in the ELIoT library (line 10). The main loop (line 13) executes periodically, with such timed behavior realized with local timer messages (line 14). When such a message arrives, the process queries the sensing device (line 17) [2] and broadcasts the value to all reachable nodes using the **`bcast_send`** function in the ELIoT library (line 18). The receiving nodes re-broadcast the message (line 23-24) using the same function. When the message reaches a node with a specific identifier (line 20), the *guard* expression (indicated by the *when* keyword) recognizes this identifier and consequently the node treats this message differently from the others, and it unicasts the value to a given IP address, e.g., to log sensed data outside the network of embedded sensors, this time with the **`send`** function from the ELIoT library (line 21-22).

Compared to standard Erlang, the main difference in ELIoT regards the semantics of inter-node communication. Standard Erlang platforms are built atop full-fledged TCP/IP stacks ensuring reliable communication between remote processes. Such network architectures are rarely found in networks of embedded sensors and actuators, especially when communication is wireless. Communication is indeed one of the most expensive operations in such networks, e.g., in terms of energy consumption, and guaranteeing absolute reliability in such settings is usually very difficult.

As a result, we use a different syntax for ELIoT's inter-node process communication than the original Erlang syntax. In ELIoT, developers use the **`!`** sign only to communicate between local processes. Instead, remote communication is realized using specific functions from the ELIoT library, as shown in the example, whose semantics is best effort and, in wireless networks, limited to single hop. The latter choice allows developers to describe system services, e.g., routing protocols as discussed in Section IV, besides application-level functionality [2].

Using the original Erlang syntax, albeit technically possible, would trick developers to think that local and distributed inter-process communications incur the same cost, ultimately leading to inefficient implementations. Conversely, our design increases the developers' awareness of such operations, and thus allows them to reason more precisely on where and how to invest the typically scarce resources. We retain, however, the original syntax on the receiver side, which allows us to leverage the pattern matching facilities offered by the language.

The ELIoT library also supports spawning new processes to a specific node or all devices reachable using ELIoT's communication functions. These functions do not return any value; in particular, differently from Erlang, they do not return any process identifier. Indeed, such an identifier is of no use in ELIoT, since inter-node communication is based on registered names and node addresses.

B. Interpreter

Erlang was originally designed to run on embedded platforms. However, over time it has grown to include a large set of libraries and a complex run-time infrastructure, e.g., to guarantee fault-tolerance. Most of these features find limited application in embedded sensing/actuation systems, unnecessarily increasing the hardware requirements.

To address this issue, we developed a custom ELIoT interpreter by wiping off most functionality not required in our target domain. For example, we removed several libraries, e.g., Corba support, as they are not required in our target domain. Thus, our design still addresses the heterogeneity of foreseeable IoT platforms by providing

[2]The code supposes that *TEMPERATURE* is a constant referring to a GPIO pin.

an interpreted environment, but also drastically reduces the hardware requirements, especially w.r.t. memory consumption. We hitherto tested our interpreter on two platforms: *i)* an ARM-based board with 64 MB of RAM and 64 MB of flash memory, and *ii)* a board with an RT3050 chip coupled to a MIPS processor, 32 MB of RAM and 8 MB of flash, featuring both wireless and wired networking.

With the current prototype, running a sensor network collection protocol, as reported Section IV, consumes about 5MB of RAM and uses very few CPU cycles (less than 5%) on the ARM board. Nevertheless, we are working on further reducing the footprint by modifying the interpreter itself and by further removing processes and features unlikely to be used in IoT embedded sensing scenarios. The ELIoT interprocess communication library, described in Section III-A, goes exactly in this direction. Indeed, not only it adapts the communication model to the peculiarities of the target platforms, which hardly implement full fledged TCP/IP stacks, but it also represents the pre-requisite to remove all the standard inter-node communication mechanisms of the original interpreter.

C. Simulator

Debugging and testing embedded sensing/actuation systems is a key area scarcely supported by most programming platforms. Gaining the required visibility into the system state, in particular, is deemed to be a key issue [4]. ELIoT offers a great opportunity to overcome this situation. By leveraging Erlang's blurred distinction between local and distributed functionality we developed a custom simulator that allows:

- to simulate an entire system by instantiating a set of virtual nodes running *unmodified* ELIoT code;
- to model communication between nodes according to *real* wireless traces for increased fidelity[3];
- to *interact* with the simulation, if required, via a standard Erlang shell, e.g., to proactively inject messages;
- to run a *mixed deployment* where virtual nodes can seamlessly interact with physical devices[4].

The simulator thus allows to start debugging a system in a fully simulated deployment, and then progressively move to a setting where the execution also spans physical nodes. This retains visibility into the system state through the simulated nodes, but it also allows to check the execution of real hardware and the interactions with the physical environment. As example, one could validate the operation of an embedded sensing application by employing real devices to check its sensitivity to sensor data, which is typically hard to simulate, and still use the simulated nodes to test the operations of the underlying routing protocols. All this happens with the guarantee that the code being tested coincides, line by line, with the code that developers will deploy.

[3]We use the traces from the TOSSIM simulator [12], although using different traces would only require developing the needed model translation.

[4]The current prototype supports mixed deployments only with hardware devices that provide an Ethernet or WiFi connection, but nothing precludes supporting other networks, like 802.15.4, provided the PCs running the simulator instances can access such networks, e.g., via an ad-hoc gateway.

```
1   trickle(Tau, {TauRef, TRef}, Counter, Version) ->
2     receive
3         transmit when Counter<2 -> ...
4         transmit -> ...
5         restart -> ...
6         {SourceId, RSSI,
7          {<<NewSrc:16/unsigned-little-integer,
8             NewVersion:32/unsigned-little-integer,
9             NewPayload/binary>>}
10        } when NewVersion>Version -> ...
11        {SourceId, RSSI,
12         {<<NewSrc:16/unsigned-little-integer,
13            NewVersion:32/unsigned-little-integer,
14            NewPayload/binary>>}} -> ...
15    end.
```

Figure 3. ELIoT implementation of Trickle (excerpt).

IV. Evaluation

To evaluate the expressiveness and effectiveness of ELIoT we consider three distributed protocols representative of typical embedded sensing/actuation applications: an opportunistic flooding protocol [13], the Trickle protocol for data dissemination [14], and the Collection Tree Protocol (CTP) [15]. The first protocol (part of CCBR [13]) addresses the broadcast storm problem [16] by using re-transmission timers proportional to the RSSI of the received message. Trickle is a widely adopted solution to propagate data (e.g., code or operating parameters) in sensor networks. In our implementation, we propagate an opaque payload. CTP is the most complex and serves to transport data from all nodes to the closest data sink according to a routing metric.

ELIoT in practice. Figure 3 shows the core part of the ELIoT code for Trickle. We use the last two matches (lines 6 to 9 and 11 to 14) to receive a message from a different node. The first parameter is added by the network layer, and includes the sending process identifier and message RSSI. The third parameter is the message itself, which is explicit for pattern matching here, while in Figure 2 was seen as a single variable. The code highlights the use of guards to improve readability. The last two matches separately specify the actions to take when a new version message has been received or otherwise. Such distinction depends on a message field (**`NewVersion`**), accessed by pattern matching the appropriate portion of the bit sequence. These can generally be used to pattern match a binary "blob" by decomposing it in fields, each with its own length and type. For example, the **`NewVersion`** field is specified as a 4-byte little-endian unsigned integer (line 8). This caters for an elegant mechanism to concisely describe message parsing.

Leveraging the actor-like model concurrency model, to implement CTP we use four processes to modularize the functionality: the *main* process is in charge of receiving messages from other nodes; the *link estimation* process

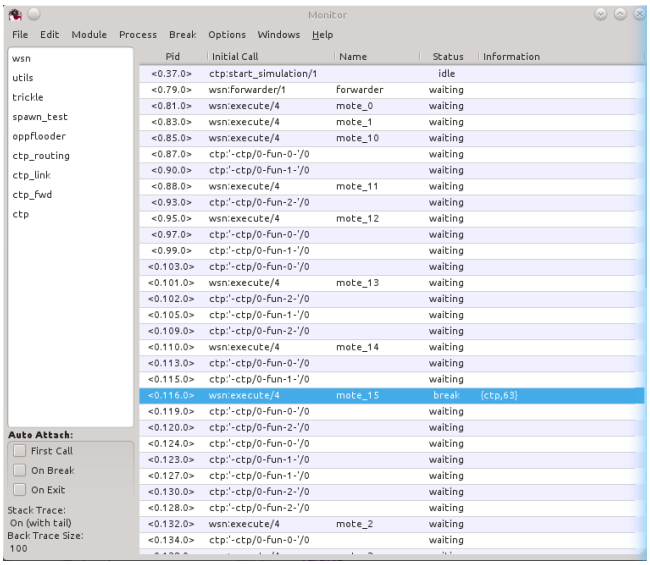

Figure 4. ELIoT processes monitor.

computes routing metrics to the closest sink and report changes to the *routing* process. This, in turn, provides the *forwarding* process with the most efficient route to use. Splitting functionality results in more readable code, with a correct separation of concerns, making the individual processes more reusable. Moreover, the hot-swapping capability can be used for changing only one part of the protocol, e.g., to replace the link estimation protocol with a different one.

Figure 4 is a snapshot of the *processes monitor* interface while running CTP. The process monitor allows to inspect each process, showing variable values and letting developers insert breakpoints into the code. The screenshot refers to a mixed network configuration, enabled by our simulator: we use a laptop running 14 simulated nodes, a more powerful desktop running 48 simulated nodes, and two real ARM devices. The possibility of inspecting the state of simulated nodes using the process monitor, while they interact with the real devices allowed us to easily spot bugs.

Comparison. We were also interested in measuring to what extent ELIoT reduces the programming effort. To this end, we measure the uncommented lines of code as an approximate measure of the coding effort and we compare the ELIoT implementations of the aforementioned protocols with their counterparts in TinyOS [10] and Contiki [11].

Figure I reports the values we measured. It shows that the more complex is the protocol, the greater is the advantage in using ELIoT. Indeed, while Contiki already provides a (slight) advantage over TinyOS, ELIoT shows even greater improvements, due to the functional and declarative model. This, together with the powerful pattern matching and serialization/deserialization features, results in implementations

Table I
PROTOCOL IMPLEMENTATIONS LINES OF CODE.

Algorithm	TinyOS	Contiki	ELIoT
Opportunistic flooder	495	187	100
Trickle	219	194	61
CTP	2169	1470	303

up to seven times more compact than their TinyOS and Contiki counterparts.

V. Related Work

Solutions exist to develop applications spanning both standard Internet devices and networks of embedded sensors and actuators. As example, the CoAP framework provides a RESTful interface on resource-poor devices [17]. Implementations of such framework exist for common sensor network platforms [18]. Unlike our approach, however, the application logic runs entirely outside the embedded network, whereas sensor and actuators are essentially application-agnostic. This spares the need for embedded system programming at the price of reduced performance. Differently, we design ELIoT to allow developers to deploy even the entire application logic on embedded devices, while still retaining a high-level programming model.

The operating system facilities are the most used programming platform in sensor networks. However, the low level of abstraction typically provided usually results in entangled implementation that are difficult to debug and maintain [2]. In this setting, TinyOS [10] and Contiki [11], which we use in the comparison of Section IV, are most often employed. Commercial products may also come with their own platform-specific APIs [19]. Applications developed atop these APIs, however, are often difficult to port to different platforms.

Higher-level sensor network programming abstractions, on the other hand, often sacrifice generality for simplicity [2]. For example, Regiment [20] features a functional programming model, providing primitives such as *fold* and *map* to process data originating from subsets of nodes. Flask [6] provides a data-flow programming model based on discrete computational steps, akin to side effect-free function calls. Snlog [21] is a rule-oriented approach inspired by logical programming, where rules are recursively applied on data available in a dedicated repository. Common to these approaches—and in fact to most solutions in the field [2]—is that the language constructs are compiled to TinyOS or Contiki code before deployment. Thus, the code running on the embedded device bears little resemblance to the hand-written one, complicating testing and debugging.

ELIoT sits in the middle between an operating system layer and higher-level programming solutions. It still allows the implementation of both system- and application-level functionality, yet it spares programmers from many low-level details and the intricacies of OS-level scheduling. The latter is mainly due to the actor-like concurrency model,

which also helps dealing with distribution by blurring the distinction between local and remote processes. Its interpreted nature also maximizes code portability, an asset in the heterogeneous IoT scenarios.

Finally, worth noticing is that sensor network operating systems often come with an accompanying simulator, e.g., TOSSIM [12] for TinyOS and Cooja/MSPSim [22] for Contiki. This is key to quickly prototype applications and has often significantly contributed to the adoption of the platform. We do the same with ELIoT, with the added feature of enabling mixed deployments with some simulation instances running on real devices, akin to EmStar [23].

VI. Conclusion and Future Work

We presented ELIoT, an Erlang-based development framework for IoT embedded sensing/actuation systems. Erlang's distinguishing features, such as the native support for concurrency and distributed programming by virtue of the actor-like model, played as stepping stones for ELIoT. To address the specific requirements at stake we designed and implemented a custom library for programming sensing/actuation embedded systems and a dedicated run-time support meeting the resource constraints of typical IoT devices. To aid in testing, we also provided a dedicated simulator able to run mixed deployments of simulated nodes and real devices. Our experience in implementing three sensor network distributed protocols demonstrated the advantages in terms of readability of code and maintainability.

Our immediate research agenda includes further investigation to reduce the hardware requirements of the ELIoT interpreter, possibly by rewriting it from scratch. We also aim at providing a dedicated sensor API to interact with sensing devices, and leverage existing work in formal verification of Erlang code [24] to enable exhaustive validation of ELIoT implementations.

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

[1] F. Kawsar, G. Kortuem, and B. Altakrouri, "Supporting interaction with the internet of things across objects, time and space," in *Proc. Internet of Things Conference*, 2010.

[2] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, 2011.

[3] G. P. Picco, "Software engineering and wireless sensor networks: happy marriage or consensual divorce?" in *Proc. FSE/SDP Workshop on Future of Software Engineering Research*, 2010.

[4] A. Bernauer and K. Roemer, "Meta-debugging pervasive computers," in *Proc. Workshop on Programming Methods for Mobile and Pervasive Systems*, 2010.

[5] M. Hossain, A. Alim Al Islam, M. Kulkarni, and V. Raghunathan, "uSETL: A set based programming abstraction for wireless sensor networks," in *Proc. Int. Conf. on Information Processing in Sensor Networks*, 2011.

[6] G. Mainland, G. Morrisett, M. Welsh, and R. Newton, "Sensor network programming with Flask," in *Proc. Int. Conf. on Embedded Networked Sensor Systems*, 2007.

[7] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proc. Int. joint Conf. on Artificial intelligence*, 1973.

[8] F. Mattern, T. Staake, and M. Weiss, "ICT for green: how computers can help us to conserve energy," in *Proc. Int. Conf. on Energy-Efficient Computing and Networking*, 2010.

[9] J. Ko, C. Lu, M. Srivastava, J. Stankovic, A. Terzis, and M. Welsh, "Wireless sensor networks for healthcare," *Proc. IEEE*, 2010.

[10] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "TinyOS: An operating system for sensor networks ambient intelligence," in *Ambient Intelligence*, 2005.

[11] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proc. Int. Workshop on Embedded Networked Sensors*, 2004.

[12] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire TinyOS applications," in *Proc. Int. Conf. on Embedded Networked Sensor Systems*, 2003.

[13] G. Cugola and M. Migliavacca, "A context and content-based routing protocol for mobile sensor networks," in *Proc. European Conf. on Wireless Sensor Networks*, 2009.

[14] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *Proc. Conf. on Symposium on Networked Systems Design and Implementation*, 2004.

[15] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection Tree Protocol," in *Proc. Int. Conf. on Embedded Networked Sensor Systems*, 2009.

[16] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu, "The broadcast storm problem in a mobile ad hoc network," in *Proc. Int. Conf. on Mobile Computing and Networking*, 1999.

[17] Z. Shelby, K. Hartke, C. Bormann, and B. Frank, "Constrained application protocol (CoAP)," `draft-ietf-corecoap-07`, 2011.

[18] M. Kovatsch, S. Duquennoy, and A. Dunkels, "A low-power CoAP for Contiki," in *Proc. Int. Conf, on Mobile Ad-hoc and Sensor Systems*, 2011.

[19] "Waspmote," `www.libelium.com/waspmote`.

[20] R. Newton, G. Morrisett, and M. Welsh, "The Regiment macroprogramming system," in *Proc. Int. Conf. on Information Processing in Sensor Networks*, 2007.

[21] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica, "The design and implementation of a declarative sensor network system," in *Proc. Int. Conf. on Embedded Networked Sensor Systems*, 2007.

[22] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón, "COOJA/MSPSim: interoperability testing for wireless sensor networks," in *Proc. Int. Conf. on Simulation Tools and Techniques*, 2009.

[23] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin, "EmStar: a software environment for developing and deploying wireless sensor networks," in *Proc. USENIX Annual Technical Conference*, 2004.

[24] Q. Guo, J. Derrick, C. B. Earle, and L.-A. Fredlund, "Model-checking Erlang: a comparison between EtomCRL2 and McErlang," in *Proc. Int. Conf. on Testing - Practice and Research Techniques*, 2010.