POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

ELIoT:
A Programming Framework for the Internet of Things

Doctoral Dissertation of:
Alessandro Sivieri

Advisor:
Prof. Gianpaolo Cugola

Tutor:
Prof. Luciano Baresi

Supervisor of the Doctoral Program:
Prof. Carlo Fiorini

2013 – XXVI

ABSTRACT

The world of embedded, communication-oriented devices has never been richer than in these last few years. While on one hand Wireless Sensor Networks are known and have been used for several years to monitor the environment, buildings or historical sites, the proliferation of devices such as smartphones, tablets, smart watches or music players, filled with sensors measuring and aggregating data about the user behavior or the environment in which she moves are quite new and have opened the possibility of developing applications in novel scenarios, such as air quality monitoring, energy distribution measurements, NFC payments, Machine-2-Machine interactions and so on.

The spectrum of devices that enable this behavior goes from the tiniest microcontrollers, such as WSN motes with 8/10 KBytes of memory, to smartphones and tablets with multicore processors and GBytes of disk and RAM. Application development methodologies for these devices have some common characteristics: from the use of particular languages and programming paradigms to the difficulties of testing and debugging applications before deployment.

There are also profound differences: Wireless Sensor Network research focuses on the development of algorithms, applications and power-saving protocols, where only localized interactions between the devices (sensors/actuators) are conceived, with small or no communication with the external world (e.g., Internet, cloud systems, remote computers); frameworks for more powerful devices focus on the ease of development and deployment of applications, with rich interactions with the Internet and with remote systems in charge of elaborating the collected data, often neglecting possible local interactions between devices located within communication range with each other.

Aim of this thesis is to show that a different route is possibile, which allows the development of applications covering a significant part of the spectrum described above, allowing **localized and remote interactions**, using high-level languages apt to manipulate data and protocols, offering powerful tools to debug, test and verify applications before their deployment.

This thesis will present a new framework called ELIoT (ErLang for the Internet of Things), that follows this new route by adapting the Erlang programming language to the requirements described above, providing a Virtual Machine and language constructs apt to the development of distributed mobile applications on embedded devices.

ELIoT also provides tools to the developers, in particular a simulator and a model checker, to debug and verify the applications before deployment, a necessary step in these scenarios, where a live monitoring/debugging of the functionality in the environment can be difficult and expensive.

This thesis will show the impact of using ELIoT in terms of productivity for developers and performance of applications developed in typical "Internet of Things" scenarios.

SOMMARIO

Il mondo dei dispositivi integrati in grado di comunicare tramite reti wireless non è mai stato così ricco come in questi ultimi anni. Mentre da un lato le Reti di Sensori Wireless (Wireless Sensor Networks) sono ormai note e sono state usate per diversi anni per monitorare l'ambiente, gli edifici ed i loro impianti o siti di rilevanza storica, dall'altro è piuttosto recente la proliferazione di dispositivi come smartphone, tablet, orologi "intelligenti" o riproduttori musicali, ciascuno contenente numerosi sensori che continuamente misurano ed aggregano dati riguardo il comportamento dell'utente o dell'ambiente in cui si muove; questo ha aperto nuove possibilità ed introdotto nuovi scenari in cui sviluppare applicazioni, come ad esempio la misurazione della qualità dell'aria a basso costo nelle città, il monitoraggio della distribuzione di energia, il pagamento elettronico con qualunque dispositivo (e.g., NFC), interazioni automatizzate tra macchine (Machine-2-Machine) e molto altro.

Lo spettro di dispositivi che permettono questo è molto ampio e va dai più piccoli microcontrollori, come i nodi wireless da 8/10 KByte di memoria, agli smartphone e tablet con processori multicore e diversi GByte di memoria. Lo sviluppo di applicazioni per questi dispositivi ha alcune caratteristiche comuni: dall'uso di linguaggi e paradigmi di programmazione specifici alle difficoltà nel provare e risolvere eventuali errori prima di installare l'applicazione nel suo contesto reale.

Allo stesso tempo ci sono anche profonde differenze: la ricerca nell'ambito delle reti di sensori si concentra sullo sviluppo di protocolli che permettano un risparmio energetico significativo, su algoritmi ed applicazioni dove le comunicazioni rilevanti sono essenzialmente locali (verso sensori ed attuatori), e la comunicazione verso il mondo esterno (e.g., Internet, sistemi cloud, computer remoti) è pressochè inesistente; i framework per dispositivi più potenti si concentrano invece sulla facilità nello sviluppo e nell'installazione di applicazioni aventi ricche e frequenti interazioni con Internet e sistemi remoti, i

quali si occupano di elaborare i dati raccolti, spesso ignorando le interazioni locali tra dispositivi.

Scopo di questa tesi è mostrare come sia possibile perseguire un modello differente, che permetta lo sviluppo di applicazioni per un'ampia parte dello spettro descritto in precedenza, e che faciliti la realizzazione di comunicazioni **sia locali che remote**, che utilizzi linguaggi di programmazione ad alto livello in grado di manipolare facilmente dati e protocolli, che offra utili strumenti di debug, test e verifica di applicazioni prima del loro rilascio.

Questa tesi introduce un nuovo framework chiamato ELIoT (ErLang for the Internet of Things), che implementa questo modello adattando il linguaggio di programmazione Erlang ai requisiti descritti in precedenza, fornendo una macchina virtuale e costrutti del linguaggio atti allo sviluppo di applicazioni distribuite per dispositivi mobili.

ELIoT inoltre fornisce strumenti agli sviluppatori, in particolare un simulatore ed un model checker, per verificare l'applicazione prima del suo rilascio, una funzionalità essenziale in questi scenari, dove il monitoraggio o il debugging dell'applicazione una volta operativa può essere difficile e costoso.

Questa tesi mostrerà l'impatto dell'usare ELIoT in termini di produttività per gli sviluppatori e di prestazioni dei programmi sviluppati in scenari tipici della cosiddetta "Internet delle Cose" ("Internet of Things").

RINGRAZIAMENTI

```
#include <triennale/ringraziamenti.h>
#include <specialistica/ringraziamenti.h>
```

Ammetto di averci pensato un po' su, questa volta, prima di accingermi a scrivere queste poche pagine intitolate, come sempre, "Ringraziamenti". E' stato più un pensiero che si aggirava in fondo alla testa, e non qualcosa di veramente cosciente, e la struttura finale di questa sezione, che state leggendo ora, è un'idea che credo di aver avuto questa primavera (per intenderci: inizio a scrivere tutto ciò a fine giugno...). In sostanza, trovavo abbastanza inutile star lì a riempire tre pagine con nomi e cognomi di tutte le persone che ho conosciuto in questi tre anni, per non parlare naturalmente di chi conosco da 10 o da 20 o da una vita intera: ognuno ha avuto un proprio ruolo in questo mio percorso, ed ognuno avrebbe diritto ad essere ricordato ad un certo punto. Ho incluso (*pun* intended) i ringraziamenti delle tesi passate, e devo comunque ringraziare da un lato la mia famiglia, che ha sempre supportato ogni mia decisione, e dall'altro il Prof. Carlo Ghezzi, che mi ha dato la possibilità di fare questo dottorato, il Prof. Gianpaolo Cugola, mio advisor, ed il Prof. Luca Mottola, co-autore di tanti articoli in questi tre anni: tutti e tre sono stati punti di riferimento per la loro amicizia ed i loro consigli.

Preferirei invece passare il resto di queste poche pagine a parlare del dottorato in sé: mi piacerebbe infatti dare un parere conclusivo, per così dire, a questa mia esperienza, e questo perchè ogni esperienza di dottorato è molto diversa dalle altre, dipende da moltissimi fattori collegati tra loro (Sei una persona autonoma? I tuoi collaboratori preferiscono una persona autonoma? La tua tesi ha senso per la comunità in cui lavori? Qualcun altro sta provando le tue stesse teorie? Etc etc etc), e di conseguenza vale la pena sentire tante campane diverse, semmai si stesse pensando se affrontare questo percorso oppure per avere un'idea di che cosa questa esperienza può dare.

Se dovessi riassumere tutto in una frase, direi: "sì, lo rifarei tutta la vita", e per tanti motivi. Il primo è dovuto alle persone incontrate: il gruppo in cui mi sono trovato è un gruppo di persone in gamba, in grado di coinvolgere gli altri nel proprio lavoro e con una grande voglia di provare cose nuove, sperimentare idee e tecnologie, senza in alcun modo farti sentire "l'ultimo arrivato" nel tuo ambito di ricerca. Da un lato mi rendo conto, forse, di trovarmi in un'università che sembra risentire di meno della situazione a dir poco ridicola della ricerca in questo Paese, e che riesce ad offrire ai suoi studenti opportunità sicuramente notevoli, dall'altro ogni nuova "categoria" di persone che incontro in questi anni al Politecnico riesce sempre a stupirmi positivamente, ed a farmi amare questa università. La possibilità di sentirti ingegnere fino in fondo, collaborando con persone che hanno una preparazione completamente diversa dalla tua ed in cui ciascuno mette a disposizione il proprio sapere per ottenere un risultato concreto. Il secondo motivo è legato al fatto che, come mi era già accaduto in passato, mi sono ritrovato a fare cose che non avrei mai pensato di fare, e che mi sono tutto sommato piaciute: tre anni fa mi ero detto che mai avrei messo le mani in una macchina virtuale esistente (paginate e paginate di codice C), ed invece l'ho fatto, con risultati tutto sommato positivi, come potrete leggere nei capitoli successivi. Altre cose fatte in questi anni in realtà hanno riflettuto passioni che sono nate nel tempo, ad esempio questa ripresa del lato "elettronico" della vita, che credevo fosse stato archiviato in un cassetto una volta finita la laurea. Un altro motivo è il fatto che, in questi tre anni, credo di essere diventato un informatico migliore. Si dice, ed è vero, che il dottorato ti spinge in una direzione ben precisa, nello sfondare quella sfera di conoscenze (ricordando una bella metafora visiva che era descritta in un blog qualche tempo fa) e nel "creare" qualcosa di nuovo. Ebbene, il dottorato permette anche di allargare la propria conoscenza in altri campi della tua materia, e permette di apprezzare cose che in precedenza non si apprezzavano allo stesso modo. Ora, non so se questo sia legato al fatto che si iniziano ad usare "sul serio" gli strumenti che si possiedono, o al fatto che la laurea non insegna o non trasmette con sufficienza questo tipo di messaggi (e temo vivamente che questa seconda idea non sia così lontana dalla

realtà), ma ho dovuto aspettare dieci anni per apprezzare il C e la sua gestione manuale della memoria e dei tipi; ho dovuto aspettare dieci anni per capire come (solo?) l'immutabilità dei dati abbia un senso nell'informatica odierna; ho dovuto aspettare dieci anni per sentire la mancanza di quell'esame di elettronica che mi ha fatto penare non poco a suo tempo. Un altro motivo sono le opportunità internazionali offerte, di cui io ammetto di aver approfittato poco (qualcuno ha detto conferenze solo in Svizzera?), ma che sono lì e aprono tutto un mondo diverso in cui però ti senti sempre a casa (tutta l'università è paese). La possibilità di conoscere ed incontrare persone di cui hai letto libri, visto filmati e che improvvisamente sono lì davanti a te a discutere di argomenti che capisci veramente; il poter visitare realtà diverse dalla tua; il poter presentare il tuo lavoro ad altri e scambiare idee ed opinioni con persone che si occupano esattamente degli stessi problemi di cui ti stai occupando anche tu.

Potrei continuare e scrivere altre motivazioni, ma direi che queste sono le principali che mi hanno fatto apprezzare a fondo questi tre anni di esperienza, che rifarei da capo a piedi, magari non proprio uguali ma sicuramente con lo stesso spirito e la stessa voglia di apprendere cose nuove, sperimentare cose nuove, magari trovare un'idea che possa, in un modo o nell'altro, cambiare il mondo.

Siv

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The world of embedded, communication-oriented devices has dramatically changed in the last few years, bringing new challenges and opportunities to researchers and developers of embedded, distributed and integrated systems: from messaging protocols to tools and facilities for the developers, to completely new scenarios enabled by this *device revolution*.

1.1 THE DEVICE REVOLUTION

Ten years ago, this world was dominated by Wireless Sensor Networks (WSNs): networks of very simple and low power devices, able to communicate with each other and to interface with sensors and actuators to collect data from the environment and interact with it accordingly. If a user needed a sensing application, she had to design her own hardware board with microcontrollers and integrated circuits, and to develop the application from scratch, using low level frameworks accessing the hardware almost without abstractions.

In these ten years, several key technological and methodological changes started a revolution in the consumer embedded world:

- microprocessors have become cheaper and more power efficient such that they can easily be integrated in mobile devices, which in turn have become more powerful and able to run applications that were once reserved for standard computers
- Web technologies have improved greatly (the so-called "Web 2.0" before, and cloud computing now), and new applications able to collect, analyze and show data dynamically ("online" with respect to "offline" visualization) are now possible and can be run on any browser, regardless of the capabilities of the underlying hardware

- as a consequence of the previous two innovations, the market has been flooded by a whole new world of mobile devices, able to run complex applications, collect data from a variety of integrated sensors, and combine these data to produce actions and to try to understand the environment in which they are located and the behavior of their owners
- the embedded world has also changed: the "Open Source Hardware" movement allows the exchange of projects and schematics of electronic devices and boards and the software that runs them, and projects such as the *Raspberry Pi* have lowered the barrier of putting together microprocessors, sensors and actuators and allowed hobbyists to build applications that collect data from the environment and react to it, at a much lower cost with respect to the ten years ago world
- the so-called "Internet of Things" phenomenon started building up, as a consequence of availability of sensing and actuating devices powerful enough to be programmed as standard computers and to be integrated into the Internet, and to interact with the users through intuitive and dynamic interfaces
- finally, research in the WSN and embedded fields have produced reliable algorithms to be used in these applications, enabling low powered, wireless and distributed scenarios.

For example, the Apple iPod Nano, a small commercial music player designed to be used while doing physical exercises, is able to collect data from an accelerometer (and other sensors) and trace the distance and speed that the owner is travelling, and send these data to a Web site that will aggregate it and show several statistics to the user. Another example is the *OpenEnergyMonitor* [1] initiative, which aims at creating a completely open source (software and hardware) platform that measures the energy consumption of domestic or industrial machinery and buildings, which is needed to make people aware of the energy sustainability problem. The hardware can be ad-hoc or rely on boards such as the *Raspberry Pi*, and the software comprises a

Web application collecting and aggregating data, and showing it to the user in a very intuitive way, using the technologies mentioned before.

1.2 PROGRAMMING FRAMEWORKS

As software engineers, we are interested in how these devices can be programmed, which tools are given to the developers to write applications that can comprise a wide variety of hardware, from the tiniest boards used in Wireless Sensor Networks, with 8/10 KBytes of memory and usually powered by small batteries, to smartphones and tablets, with powerful processors and GBytes of memory, powered by bigger batteries or (in some cases) power supplies.

The programming frameworks that can be used in these applications have some common characteristics: they target a usually small part of the device spectrum; in the case of WSN the hardware abstractions typical of the most famous operating systems, TinyOS [2] and Contiki [3], have been conceived to support the boards available today, where a profound knowledge of the harware is needed by the developer in order to port an application to a different platform; the situation is similar for hobbyist platforms such as Arduino. Smartphone and tablet frameworks may use the same programming languages used for computer applications, but there can be differences in how the same API behaves on these devices, and the control flow of the application is quite different than a computer application, essentially due to the multitasking peculiarities of such devices. Boards such as the *Raspberry Pi* are the most similar to a computer from the developer's point of view, and only the low level hardware interfaces are platform specific.

Difficulties in testing and debugging the applications are also common for these scenarios: this is often due to the unpredictability of at least part of the interactions within the environment, that will be discovered only when the application is actually deployed; smartphone applications also rely quite heavily on Internet connections, and their behavior can become erratic if the connection is faulty, condition that can happen in several situations.

At the same time, there are profound differences: on the one hand, WSN research focuses on algorithms to collect data efficiently and to save as much energy as possible, giving the developer the programming tools to strongly tailor the application to her specific requirements (e.g., need for more communication reliability). These frameworks also consider interactions between devices to be *local*: data is exchanged inside the same network and decisions are taken locally. Interactions with the Internet are not common, since these devices have such low capabilities that it becomes quite difficult to add data exchange with the external world, and often the only external link is made by the sink, which is usually a computer or a more powerful device.

On the other hand, frameworks for more powerful devices focus on the ease of programming, to keep the barrier low to new developers, and the devices are usually in charge of collecting data and sending it to remote servers, often located in today's cloud systems, which will elaborate these data and show it to the user, usually through a Web site (e.g., the music player example cited before). Local interactions are usually neglected and devices rarely communicate with each other, even if they are in communication range and integrate more powerful communication capabilities, different communication media, and more energy and power capabilities than, for example, WSN.

1.3 ELIOT: THE THIRD ROUTE

The aim of this thesis is to show that a different route is indeed possible: provide a framework that allows developing applications including both *local and remote interactions*, using a high level language with respect to some existing frameworks, offering tools that allow testing and debugging the applications before deployment, simulating tens or hundreds of devices and at least some of the characteristics of the environment. At the same time, such a framework would cover a great part of the device spectrum, at worst with acceptable compromises for the less powerful devices.

This thesis will present such new framework called ELIoT (ErLang for the Internet of Things) [4] [5] that follows this route, adapting an existing pro-

gramming language (Erlang) to the requirements and needs of the embedded communication-oriented world, leveraging the existing capabilities of the language and virtual machine, providing an abstraction over the hardware on which the applications will be executed, introducing new language constructs apt to the so-called "Internet of Things" world.

The framework will also provide developers with tools for testing, debugging and verifying their code: a simulator that allows executing the application on a simulated environment without changing a single line of code, static analysis tools already provided with the Erlang libraries and compatible with ELIoT, and a model checker that verifies properties of the application being developed. Other virtual machine capabilities are leveraged to allow *hot swap* of code and live interactions with the running system.

This thesis will then present a possible scenario where the framework can be applied, and it will show the impact of using ELIoT in terms of productivity and performances, precisely identifying the trade-offs between ease of development, access to low level hardware and choices left to the developer, based on her experience and application requirements.

1.4 STRUCTURE OF THE THESIS

This thesis is structured as follows:

- Chapter 2 will describe the foundations of the ELIoT framework, in particular the Erlang programming language and the REST framework that is offered by ELIoT to access devices and sensors through the Internet
- Chapter 3 will introduce the scenario that we used for evaluating our platform: it will be used as a running example when showing the ELIoT features, and it will be a reference for performance comparison in the later chapters
- Chapter 4 will show the ELIoT architecture, the syntax and semantics differences with respect to Erlang, the changes that have been made to the virtual machine and their implementation details

- Chapter 5 will describe the the simulator and the model checker
- Chapter 6 will show the variety of hardware devices that have been tested and the qualitative and quantitative evaluations of ELIoT compared to different solutions for embedded applications
- Chapter 7 will describe in detail the existing frameworks and solutions that today's developers can use for embedded communication-oriented applications, and different choices that have been made in terms of hardware access, programming languages and patterns
- Chapter 8 will summarize the contribution described in the thesis and will present the conclusions of this work.

2

FOUNDATIONS

This chapter will describe the foundations on which the ELIoT framework is built upon: Section 2.1 will introduce the Erlang programming language, with several examples showing the main features that prompted us to base our work on this language and platform; Section 2.2 will describe the REST and Computational REST architectural styles, which have been integrated in Erlang and then used as a starting point to introduce similar features in ELIoT (see Section 4.4).

2.1 ERLANG AND THE ACTOR MODEL

Erlang is an industrial-strength functional language, which includes specific constructs to ease the development of communication protocols, data manipulation algorithms, and reliable distributed applications.

As a functional language, it contains the well known primitives and constructs that help manipulate data and write very compact code; it also supports single assignment variables and immutable data, thus avoiding side effects. As for other functional languages, there is no concept of loops: recursive functions (in particular, tail recursive functions) are used to "emulate" loops and cycle through lists or other similar data types. Even though these characteristics are important, and in Chapter 6 we will see how they impact the readability of the code, the main reason we decided to use Erlang was its support for parallel and distributed computations.

Erlang implements the actor model [6] for concurrency support: language primitives allow developers to easily spawn new processes that execute functions, and these are lightweight processes handled by the Virtual Machine, that do not correspond directly to system processes or threads; the VM sched-

```
1  % Initial function to be executed
2  start() ->
3      % Create a new process, which will execute the loop function (below)
4      Pid = spawn(fun() -> loop() end),
5      % Register its identifier, so that he can be reached by name
6      register(somename, Pid),
7      MyNumbers = [1, 2, 3, 4, 5],
8      % Send a message to that process, containing the type and the identifier
9      % of the sender ("self()")
10     somename ! {message_type_1, self(), MyNumbers},
11     % Receive the result and print it
12     receive
13         Result ->
14             io:format("~p~n", [Result])
15     end.
16
17 % Simple function returning the double of the input
18 double(Number) ->
19     2 * Number.
20
21 % Receive messages, process them, and return results to the original sender
22 loop() ->
23     % Extract the first message from the queue (blocking)
24     receive
25         % Pattern match the content of the message
26         {message_type_1, SenderPID, ListOfNumbers} ->
27             % Apply function Double to the whole list, element by element
28             Result = lists:map(double, ListOfNumbers),
29             % Send the result back to the original sender
30             SenderPID ! Result;
31         % A different content for the message
32         {message_type_2, SenderPID, Content} ->
33         [...]
34     end,
35     % Recursive call to parse next message in queue (or wait for a new message to arrive)
36     loop()
37 end.
```

Figure 1: Spawn processes and pattern match messages.

ules these processes on the different cores of the computer depending on its architecture and capabilities. Processes can share data only by communicating with each other using messages: each process has a *mailbox* associated with it, which receives messages coming from other processes, and the *send* command is a primitive of the language (as the *spawn* command is).

The example code in Figure 1 shows these characteristics: the idea is to have a process that receives a list of elements and applies a particular function to each of these elements (in the example, a simple doubling function on line 18). This process executes the **`loop`** function on line 22 recursively, and in each iteration it extracts the topmost message from its mailbox (or blocks

until a message is received) by using the **receive** primitive on line 24; the corresponding code to send messages is shown on line 10: the **!** operator takes the receiver name or identifier and sends him the message. Notably, the syntax for inter-process communication is independent of whether the communicating processes are local or remote, which simplifies distributed programming by blurring the boundary between local and remote context[1].

The command to spawn new processes is shown on line 4: the **spawn** function takes the function to be executed as a parameter, and it starts a new parallel process doing exactly that. The **register** function on line 6 takes the identifier returned in the previous line and associates a name to it, so that the spawned process can be addressed by name and not only by its identifier.

Distinguishing between message types is specified declaratively using *pattern matching*, namely, by stating constraints on the message format, as in line 26 and 32. Erlang's pattern matching also allows parsing and filtering of binary data, such as message payloads, using very compact code. This is an asset for implementing low-level communication protocols, as often required in IoT applications.

The example in Figure 2 shows how it works: lines 11-12 show how a stream of bytes is matched, and its different bits are taken apart and put into variables; in particular, the variable **Type** will contain the first 4 bits of the stream, variable **N** will take the next 12, variable **IP** takes the next 32 (an IPv4 address), and the rest of the stream is put inside a single variable that will be decoded later on; at this point, Erlang will treat all the previously mentioned variables as integers, and it will allow their manipulation.

The rest of the stream is decoded from line 34 on: in this example, we consider that a (possibly complex) Erlang term has been encoded to binary data, by using a functionality provided by the standard library, and here it is decoded and then interpreted using, again, pattern matching, this time on tuples instead of bytes.

1 An important definition, that will be used also in the next chapters, is the concept of *node*: a node in Erlang is an instance of the Erlang VM, and multiple nodes can be located on the same machine, and in different devices connected through a network.

```
% Define some constant values specifying the type of the message
-define(MSG_WITH_ACK, 1).
-define(MSG_WO_ACK, 2).
-define(ACK, 3).

receiver() ->
    % This process receives messages from a network socket
    receive
        % Bitstream: it carries the type and progressive number
        % of the message, the sender address and some content
        <<Type:4/bitstring, N:12/bitstring, IP:32/bitstring,
              Content/binary>> ->
            % Type can be immediately evaluated as an integer
            case Type of
                ?ACK ->
                    ok;
                ?MSG_WITH_ACK ->
                    send_ack(N, IP),
                    decode(Content);
                ?MSG_WO_ACK ->
                    decode(Content)
            end
    end.

% Send the ack: build the message with correct type and number,
% then send to the originating IP
send_ack(N, IP) ->
    Msg = <<ACK:4/bitstring, N:12/bitstring>>,
    send_to_ip(IP, Msg).

% Decode the content: here we are not using other binary fields
% (you can, if you really want), we are decoding the binary as an
% Erlang term, then we pattern match as before
decode(Content) ->
    case binary_to_term(Content) of
        {message_type_1, Some, Variables} ->
            perform_some_operation(Some, Variables);
        {message_type_2, Other} ->
            [...]
    end.
```

Figure 2: Binary pattern matching.

Erlang code is compiled into a bytecode, which is interpreted (or compiled just-in-time) by the Erlang Virtual Machine (VM). This provides great flexibility, allowing processes to be dynamically spawned, also across hosts, with nodes exchanging code over the network. This feature eases the dynamic (re)deployment of distributed applications: spawning a process remotely uses the same primitives as in a local setting, while the message-passing functionality remains the same because of Erlang's implementation of an overlay network of nodes, which will be described in more details later. Thus, developers

may start writing an application in a local context and then move to a distributed setting with (almost) no changes to the code. This model nicely fits massively distributed scenarios characterized by transient interactions, such as the IoT, easing software reconfiguration.

Finally, the Open Telecom Platform (OTP), part of Erlang's libraries, provides useful mechanisms to design robust distributed applications. In particular, the platform introduces several architectural/design patterns, called *behaviors*, which provide the non-functional parts of a typical architecture (such as a *server* process, handling requests) and allowing the developer to take care of the functional parts only.

Figure 3 shows a brief example of the server behavior: the developer is required to create an Erlang module implementing some specific methods (a sort of interface implementation), in particular an initialization function (line 12), which can initialize the state of the server (e.g., open database connections, open files, create data structures...), a termination function (line 17), which is called when the server is stopped, and can react to the specific reason (the server may have been stopped correctly or it may have crashed for some reason), and the functions handling the requests (lines 22 and 27), being them request-response or request-only (a peculiarity of Erlang servers) [2].

Another very important behavior is called *supervisor*, whose job is to monitor the execution of child processes and to implement the necessary failure-handling mechanisms. Supervisor processes can be hierarchically composed to structure fault-tolerant implementations according to application-specific requirements, which may come in handy for dealing with localized control loops. It is important to use behaviors whenever possible, because they integrate well with the supervisor and allow a correct failure-handling, restart of processes and *hot* code updates (more on this later).

The *supervisor* behavior is based on an Erlang mechanism called *process linking*: a process is able to spawn another process and link to it, thus receiving information (as messages) when the latter terminates its execution (whether it stops correctly or because of an error). Figure 4 shows exactly this: line 3

2 There are also another couple of functions that have to be implemented, regarding code updates and handling of extra informations, which have been omitted here for clarity.

```
1  % Specify the design pattern to be used by this module
2  -behavior(gen_server).
3  % Define a record (a named tuple) with a single field, to be
4  % the state of this server
5  -record(state, {list = []}).
6
7  % Standard call to start the server
8  start_link() ->
9      gen_server:start_link({local, ?MODULE}, ?MODULE, []).
10
11 % Initialization function
12 init([]) ->
13     {ok, #state{}}.
14
15 % Termination function for this server, called when the server
16 % is stopped or crashes
17 terminate(Reason, State) ->
18     ok.
19
20 % Handle a request that does not need a reply (the caller returns
21 % immediately)
22 handle_cast({add, Element}, State#state{list = List}) ->
23     {noreply, #state{list = [Element|List]}}.
24
25 % Handle a request that needs a reply (the caller is blocked
26 % until he gets an answer)
27 handle_call({get, head}, From, State#state{list = [Head|Tail]}) ->
28     {reply, Head, #state{list = Tail}}.
29
30 [...]
```

Figure 3: Server behavior.

allows the process to receive exit messages, line 5 spawns a new process and links to it, and line 16 matches an exit message coming from a linked process. The supervisor behavior enhances this mechanism allowing an easy creation of trees of processes, with each monitored process registered with a specific restart behavior (e.g., a process which terminates will be restarted, or it will in turn terminate its supervisor, in a chain reaction mechanism that will stop the entire application).

One last feature of Erlang is the possibility of *hot code swap*: the code can be changed on the fly, without having to stop a running application and allowing the parts executing the old code to terminate their job gracefully. This mechanism has some limitations, but it is useful at least for two reasons: it allows application updates with the application still running, and it allows

```
1  start() ->
2      % Receive a message when monitored processes die
3      process_flag(trap_exit, true),
4      % Spawn a new process and link to it (monitoring)
5      Pid = spawn_link(fun() -> loop() end),
6      register(somename, Pid),
7      {ok, Pid}.
8
9  request(Value) ->
10     somename ! {request, self(), Value},
11     receive
12         {result, Result} ->
13             Result;
14         % This message is received if a monitored process dies
15         % with the reason Reason
16         {'EXIT', Pid, Reason} ->
17             {error, Reason}
18     end.
19
20 loop() ->
21     receive
22         {request, Sender, Value} ->
23             % Assuming a hardware SPI interface is present
24             % in the platform, this function may fail
25             Res = spi:send(Value),
26             Sender ! {result, Res},
27             loop()
28     end.
```

Figure 4: Process monitoring.

the load and execution of new modules and functions dynamically, a feature that we leverage in Computational REST, as shown in the next section.

Erlang provides a stepping stone to enable development of IoT applications. On the other hand, the original Erlang's syntax, semantics, and system support are not straightforwardly applicable in IoT scenarios. The IoT communication patterns and resulting communication guarantees differ from those of traditional Erlang networks. Moreover, mainstream Erlang VMs demand hardware resources rarely found in IoT settings, whereas debugging and testing IoT applications cannot be oblivious to the real-world interactions IoT systems are exposed to. ELIoT tackles these issues as decribed in the next chapters.

2.2 REST INTERFACES AND COMPUTATIONAL REST

This section will introduce the Computational REST architectural style [7], the differences and enhancements with respect to the REST style, and the Erlang implementation that we did [8]; this implementation will then be the starting point to introduce dynamic RESTful interfaces in ELIoT (see Section 4.4).

2.2.1 *REST*

Defined by R.T. Fielding (one of the main authors of the HTTP protocol), the REpresentational State Transfer (REST) style provides an *a posteriori* model of the Web, the way Web application operates, and the technical reasons behind the Web success.

Fielding's Ph.D. thesis [9] defines the set of *constraints* that every REST application should satisfy: the structure of the application has to be client-server, communication has to be stateless, caching has to be possible, the interface of servers has to be standard and generic, layering is encouraged, and each single layer has to be independent from the others. An optional constraint suggests using code-on-demand [10] approaches to dynamically extend the client's capabilities.

These constrains are coupled with a set of *foundation principles*:

- the key abstraction of information is a resource, named by a uniform resource identification scheme (e.g., URLs);
- the representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes;
- all interactions are context-free;
- only a few primitive operations are available;
- idempotent operations and representation metadata are encouraged in support of caching;
- the presence of intermediaries is promoted.

While these principles allowed REST to be scalable and supported the current Web dimensions, at the same time not all the Web applications followed these design guidelines; for example, they might require stateful communications or they might create problems to caching devices components.

The main limitation of REST is the generic interface constraint: it improves independence of applications on specific services, because all the components are able to handle any data, but at the same time it hampers the efficiency of communication, since all data must be coded in a standard way to pass through standard, application independent interfaces; something not easy to do especially when there is more than pure "content" to be sent between peers.

The CREST authors identified this and other REST weaknesses in [11] and decided to address them by moving the focus of the communication from *data* to *computations*. If the former is the only subject of an interaction, then a client receiving a message through a generic interface could not be able to interpret it correctly. The REST optional constraint of code-on-demand is too weak to solve the issue, since the same client could not be able to use that code.

2.2.2 *Computational REST*

The result of this paradigm shift was the Computational REST (CREST) [7] style, which let *peers* exchange *computations* as their primary message, usually implementing them through continuations. These are instances of computations suspended at a certain point and encapsulated in a single entity to be resumed later. They are offered as a basic construct by some languages, usually functional ones like Scheme, which also allow continuations to be serialized and transmitted along a network connection to allow the computation to be resumed on a different node.

Whenever a language does not offer the continuation mechanism, a *closure* can be used instead: it is a function with free variables declared within its scope, and since the extent of these variables is at least as long as the lifetime

of the closure, they can be used for saving a state between different calls of the function.

Also notice that in the definitions above we used the term "peer" instead of "client" or "server". This is not by accident, since CREST does not distinguish between clients and servers but rather between *weak peers* that support a minimal subset of the CREST operations and usually operate as initiators of the interaction, and *strong peers* that support the whole set of CREST operations and characteristics and may fully interact with other peers, be they strong or weak.

CREST draws on the REST principles to define a new set of architectural guidelines:

- a resource is a locus of computations, named by a URL;
- the representation of a computation is an expression plus metadata to describe the expression;
- all computations are context-free;
- the presence of intermediaries is promoted;
- only a few primitive operations are always available, but additional per-resource and per-computation operations are also encouraged.

As for the last point, CREST defines two primitive operations: the *spawn* operation requires the creation of a process executing the computation; this process is associated to a unique URL and when this URL is invoked the computation itself is resumed and the results it produces are returned to the caller; thus, new services can be installed in a (strong) peer and then accessed by any client. The *remote* operation installs a computation and resumes it immediately, returning any result to the caller and destroying it when it ends, so that it cannot be accessed again.

In [11, 12] the authors further detail the CREST principles:

- any computation has to be included into HTTP operations, so that the new paradigm could be made compatible with the current Internet in-

frastructure. To keep up with such compatibility, the authors also distinguish between machine URLs and human-readable URLs, where the former may contain the computation itself, while the latter can be used by users;

- computations may produce different results, based on any received parameter, server load or any other factor that changes during time; they can also maintain a state between calls, for example for accumulating intermediate results;
- computations have to support independency between different calls, and avoid data corruption between parallel invocations using synchronization mechanisms offered by the languages of choice;
- computations can be composed, creating mashups: a computation may refer to other computations on the same peer or on different peers, and an execution snapshot should include the whole state of the computation;
- intermediaries must be transparent to the users;
- peers should be able to distribute computations, to support scaling and lowering latency, also checking temporal intervals between executions of the same computation and specifying some sort of expiration date when necessary.

Finally, in [12] a new feature has been introduced: spawned processes should act as so-called *subpeers*, with their own *spawn* and *remote* capabilities, inheriting security policies by their ancestors in the process tree, where the root node is the peer itself. This way a hierarchy of processes is created in a CREST peer, where each node is limited by its ancestors and limits its successors.

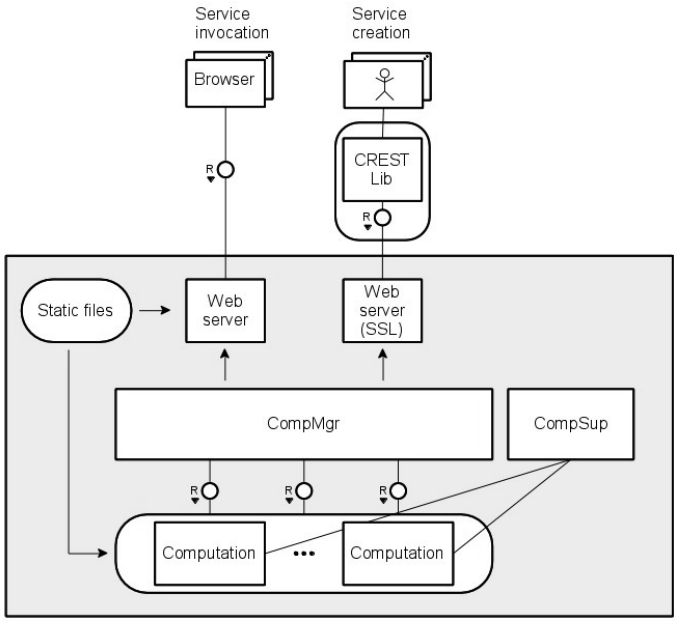


Figure 5: Server structure

2.2.3 *CREST-Erlang*

In this section we illustrate how the CREST style can be implemented in Erlang. The resulting framework is called CREST-Erlang, as opposed to CREST-Scheme, which denotes the original framework presented in [11]. Figure 5 shows the structure of a CREST peer written in Erlang. At the bottom are the computations running into the peer, which have been installed there by invoking the *spawn* or *remote* CREST primitives. They are managed by an ad-hoc component, the `CompMgr`, which installs new computations, keeps a list of those running inside the peer, and dispatches incoming invocations.

As we mentioned, one of the main reasons to choose Erlang was the support offered by the language to let (distributed) processes communicate. On the other hand, to be CREST compliant, the communication among peers has to use the HTTP protocol. Accordingly, our peer embeds a Web server, which

waits for incoming HTTP requests, unmarshals them, and uses the standard Erlang communication facilities to dispatch them to the `CompMgr`. More precisely, Figure 5 shows two Web servers, one answers HTTPS requests and is meant to handle *spawn* and *remote* operations, which we choose to securely transfer on top of SSL (more on this later). The other serves standard invocations and static pages, a trivial but required functionality for a Web framework.

As for the adopted protocol we chose, it is worth mentioning here that we decided to send computations using the HTTP POST operation, while the original CREST approach suggests embedding them into the URL of the *spawn* request. This choice seems more in line with the expected usage of HTTP. Indeed, the POST operation has been designed for those requests that are expected to alter the internal state of the receiving server, and this is the case for the installation of a new service. Moreover, the POST payload may include a large body of data, as it happens in the case of the state of a computation and the associated bytecode.

As shown in Figure 5, our framework also includes the `CRESTLib`, which provides a set of facilities to invoke local and remote services without having to bother with the underlying communication details. This is used by peer clients, but it can also be used to implement the services themselves, when they have to communicate with other peers.

Finally, to improve fault tolerance each peer is organized in a supervision tree, with a high level supervisor (not shown in figure) in charge of all the fundamental modules including the two Web servers and the `CompMgr`, and a low level one, the `CompSup`, to which all the spawned computations are attached. The former is able to monitor and restart each of its children, while the latter, at the current state, just logs any error or exception happening to computations, unlinking them from the `CompMgr` when this happens.

Listing 6 shows the template of an Erlang *service* to be spawned or remotely executed on a peer. It receives from the `CompMgr` the invocation parameters originally coming from the client, uses them to perform its computation, and finishes by invoking itself with the new state calculated during execution,

```
1  my_service(State) ->
2    receive
3      {Pid, [{"par1", P1}, {"par2", P2}, ...]} ->
4        %% Do your job accessing par1, ... parN
5        %% eventually create a new state NewState
6
7        %% If necessary, spawn myself on peer Hostname
8        invoke_spawn(Hostname, ?MODULE,
9                     fun() -> my_service(NewState) end),
10       %% Finish with a tail recursion (or just end this
11       %% computation)
12       my_service(NewState)
13   end.
```

Figure 6: Spawn a new service.

using the typical approach of functional programming based on tail recursion. Lines 8-9 show how the service may spawn a copy of itself (i.e., a copy of the computation) on a different node, if necessary.

Notice that what is transferred to the other peer through the `invoke_spawn` primitive is the *closure* of the running service, not the *continuation*, as required by CREST. Indeed, as we mentioned in the previous section, this is the only primitive offered by Erlang. On the other hand, the need to transfer computation while it is executing statements in the middle of the service's code is very uncommon. The typical service pattern is the one shown by our template, which transfers the computation just before recursing. If this is the case, transferring the closure obtains the same result as transferring the continuation of the computation.

The only point not covered by our CREST-Erlang framework is the concept of *subpeer*, which has been described by the CREST authors in a subsequent article [12], so it was not included in the current prototype.

Technologies involved and details about security

For the Web server part, we analyzed several different platforms developed in the last few years for handling HTTP communications in Erlang. Each has its pros and cons, and in the end we chose *MochiWeb* [13], because of its support to JSON (which we used to effectively serialize parameters and

Table 1: CREST implementations line code comparison

Framework	Framework source code	Demo source code
CREST-Scheme	5938	817
CREST-Erlang	2957	768

return values passed among peers and clients) and RESTful services, and for its performance.

The MochiWeb library and the OTP modules together provide the main skeleton of our peer: the supervising system, the logging system (not shown in Figure 5), and the two Web servers. This allowed us to focus on developing the functional parts of the framework.

As for security, Erlang does not offer many facilities. Indeed, it was born as a language for handling telephony devices, a domain in which security is usually guaranteed by directly controlling the network itself. Now that Erlang is being used outside its target domain, this weakness has been identified and the first security facilities are being added to the language. On the other hand, we are far from having ad-hoc facilities to manage security in general and the security of mobile code in particular. To address this issue we decided to adopt a strategy based on mutual authentication among peers. This way we bypass the specific problem of protecting the incoming computation from the peer and the peer from the computation, building a trusted network on top of which computations may roam freely. This is clearly a sub-optimal solution, which we plan to overcome in future versions of our prototype.

2.2.4 CREST-Erlang *performances*

Qualitative evaluation

To compare the effort in implementing the two framework, and so to indirectly compare the choice of the two languages used, i.e., Scheme vs. Erlang, we counted the lines of code of the main library and of the implemented case studies, not counting the external dependencies. The results are illustrated in

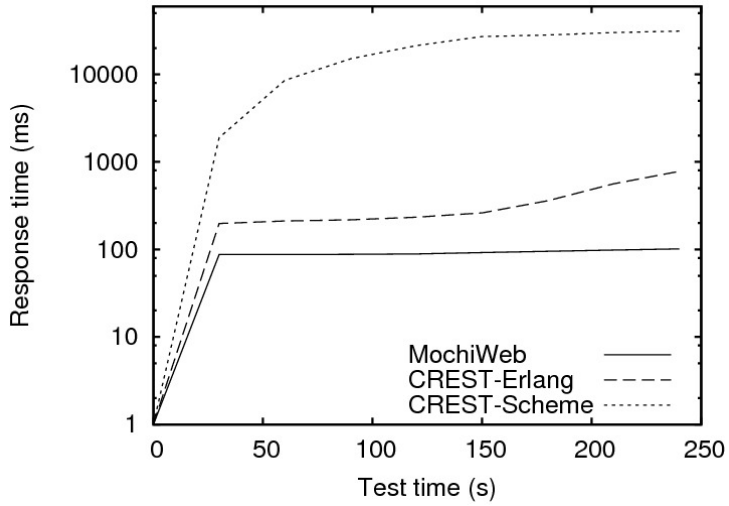
Table 1 and show that our code is about a half of the original one. This fact confirms our initial idea that Erlang more easily and naturally supports the CREST mechanisms.

Quantitative evaluation

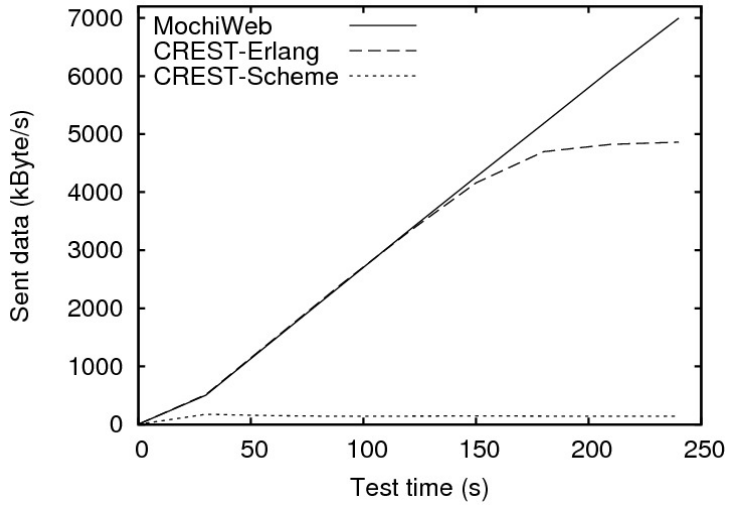
To compare the two frameworks in terms of performance, we re-built part of the implementation of the original case study, in particular we used the same Web client application (with its graphical widgets) and recreated some of the corresponding CREST services. We also implemented this case study as a standard Web application using MochiWeb alone, to use as a reference. This was possible since the original case study, unless the client duplicates its session, does not exploit any advanced CREST functionality; all computations are installed during system startup, and they are only invoked at the demo.

To actually measure the performance of these three applications, we used a dual core laptop with 4GB of RAM as a server, and we launched several simulated users from a different computer, a 6 core desktop with 8GB of RAM. Notice that we choose the machine running the clients to be more powerful than the server to be sure the values we measured were not influenced by some limitation on the client side. The two machines are connected by a 100Mbit LAN. The whole test is run by using a client application, written in Erlang, which measures the average response time for each request and the throughput in term of KBytes per second sent to the clients. We used a navigation sample recorded during a browser session through the demo site to simulate the behavior of a standard user. Through our script we simulated the arrival of one of such users every second, each repeating the same session with a delay of one second at the end, for 4 minutes in total.

Figure 7 shows the results we measured in terms of response time and throughput. The CREST-Scheme framework has the worst performances, serving a very low number of pages per second with a response time peaking at more than 30 seconds; Mochiweb performs better than CREST-Erlang in terms of response time, because of the overhead introduced internally by the latter, and it is also able to answer more requests per second in the last minute of the



(a) Response time

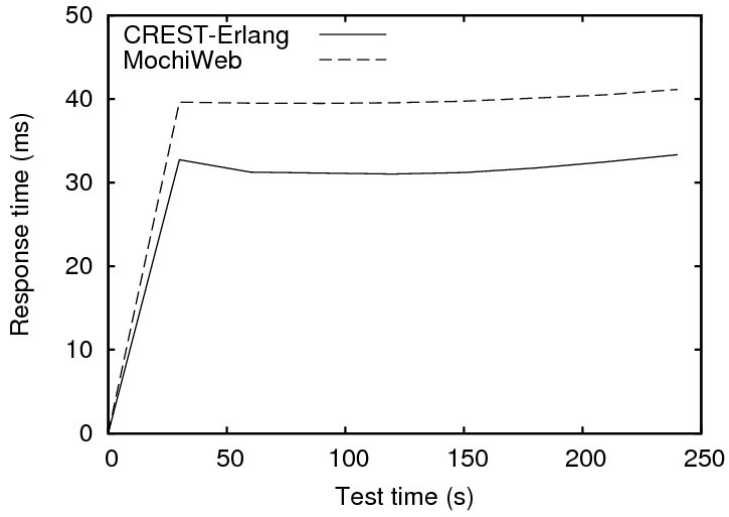


(b) Throughput

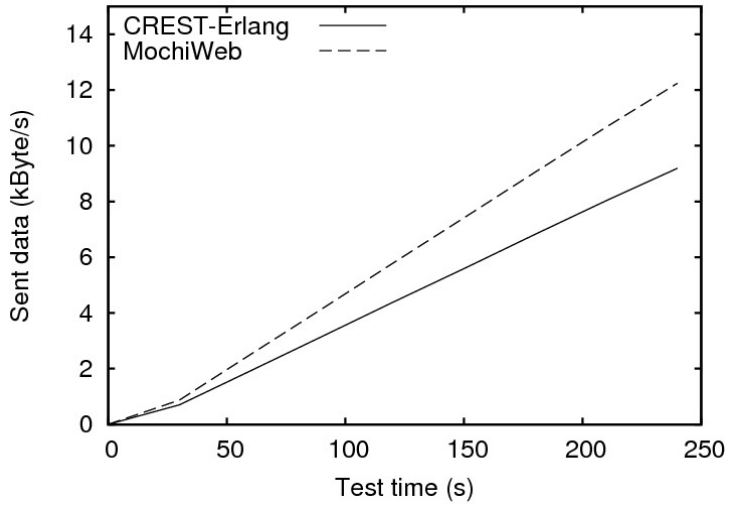
Figure 7: CREST-Scheme demo

test, because its usage of the server resources is lower than the CREST-Erlang one, especially in terms of CPU usage.

To test the overhead introduced by using the *spawn* and *remote* CREST operations, we compared our prototype against MochiWeb in running a Web application based on a simple CREST service. Each client starts by asking a front-end peer to spawn a new instance of this simple service on a different peer, located on the same machine, and from then on it invokes this new service repeatedly, with one second delay among each invocation; the MochiWeb version has the same service pre-installed, which the client invokes repeatedly as before. As in the previous case, we start one client every second for the 4 minutes of the test. Figure 8 illustrates the results we gathered in terms of response time and throughput. We notice that MochiWeb is able to answer more requests per second, and this explains the higher throughput, while the response time is similar and it remains almost constant while the number of clients increases.



(a) Response time



(b) Throughput

Figure 8: Test application

3

THE SCENARIO

This chapter will introduce the scenario we will use throughout the thesis to illustrate, validate and evaluate ELIoT.

3.1 THE SMART GRID

One of the most popular and analyzed technologies that have been under development in the last few years is the so-called *smart grid*:

> The *smart grid* is the collection of all technologies, concepts, topologies, and approaches that allow the silo hierarchies of generation, transmission and distribution to be replaced with an end-to-end, organically intelligent, fully integrated environment where the business processes, objectives, and needs of all stakeholders are supported by the efficient exchange of data, services and transactions. [14]

In particular, it is aimed to provide several improvements in terms of efficiency, reliability, sustainability:

- **reliability** gathered informations can be used to detect faults in the grid, and solutions can be found without the immediate intervention of technicians (e.g.,, multiple routes)
- **efficiency** the ability to communicate to the control panel located in each house allows utility companies to track the electricity consumption and possibly to dynamically change the amount of energy provided to prevent overloads. The possibility to inform people on their own consumptions and to calibrate electrical bills according to different policies

(again, in order to prevent peaks and overloads of the grid) is another consequence of the grid monitoring

- **sustainability** the traditional electrical grid is not built to allow many distributed feed-in points, for example renewable energy sources like *solar panels* or *windmills*, especially because these sources are often characterized by frequent fluctuations in generated power; because of this situation, informations have to be gathered from the grid to help balance these fluctuations and correctly handle the "stable" sources along these new ones, that are becoming quite popular.

The *smart grid* is an often-cited scenario for IoT applications [15]: it runs on quite different devices, from the in-house appliances to the main computers of electrical companies, and it requires both local communication (to take immediate decisions for example in case of power shortage in a neighborhood) and remote communication (to coordinate with the main power grid and to receive commands from the decision-makers).

3.2 THE IMPLEMENTED SCENARIO

The smart-home scenario we consider is an instance of the *smart grid* scenario introduced before: in particular, its base design involves the house part of the *smart grid*, which coordinates with the rest of the grid and at the same time has to take local decisions based on different parameters, both local and coming from the external network.

Base design

In general, the devices in Figure 9, being the control panel or individual appliances, need to access the Internet, e.g., the control panel must be able to obtain energy rates from the provider, and to be accessible from the Internet, e.g., appliance manufacturers must be able to remotely update the appliances' on-board software. At the same time, a local control loop, guided by the control panel, is beneficial to reduce communication costs and improve perfor-

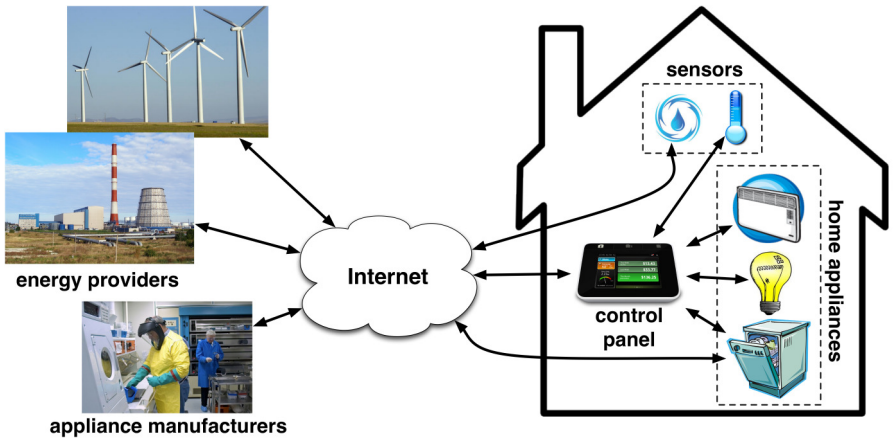


Figure 9: Smart-home application.

mances. In particular, the control panel acts as a front end for the users and coordinates the appliances' activities, dealing with:

F1: discovery and monitoring of home appliances, which provides the information to compute their operating schedules;

F2: processing of the user inputs, and computation of a schedule of appliance operation whenever required;

F3: communication with external entities, e.g., to query the energy providers for energy prices or to offer energy consumption information over the Internet.

For ease of installation, smart-home devices are expected to feature wireless communication. Because of this, we design the discovery functionality required in **F1** using a soft-state approach [16]. The control panel periodically broadcasts beacons that running appliances immediately acknowledge, either to join the system initially or to confirm their presence afterwards. In absence of acknowledgment, the control panel removes the appliance from the application state.

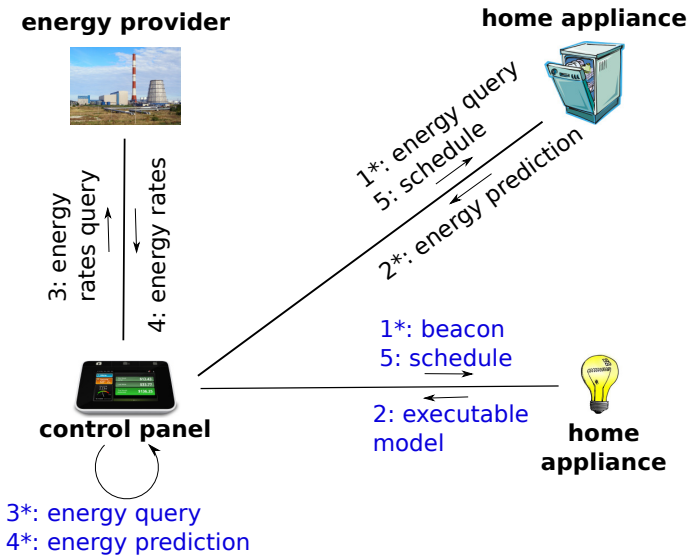


Figure 10: Scenario A and B.

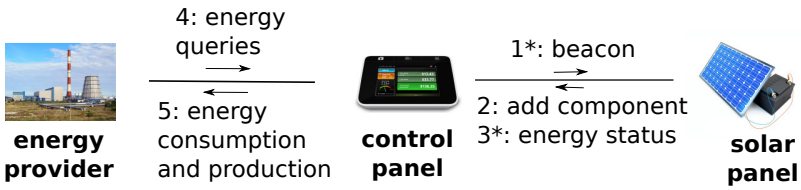


Figure 11: Scenario C.

The design of the remaining functionality depends on application requirements and available hardware platforms:

SCENARIO A: if home appliances are able to locally compute their expected energy consumption, we can design the schedule computation of **F2** by issuing remote queries to obtain the corresponding information. This is shown in the black sequence of exchanges in Figure 10: whenever the user inputs new information, the control panel queries the appliances for their expected energy consumption according to different operating

settings (step 1 and 2), and asks the energy provider for the energy rates at different times of the day (step 3 and 4). Based on this and environmental data collected from sensors, the control panel distributes an operating schedule back to the appliances (step 5). The algorithm that implements this behavior is shown by Algorithm 1: it is a formalization of this sequence of steps, and it has been implemented in the smart-home application that we developed and that will be described in the next chapters

SCENARIO B: if an appliance's computational power is severely constrained, e.g., in the case of a light fixture, or the amount of data to exchange is excessive, the estimation of expected energy consumption for **F2** should be performed by the control panel itself. The blue sequence of message exchanges in Figure 10 illustrates a design supporting this form of interaction, which requires computationally-constrained appliances to provide the control panel with a model of their expected energy consumption. The light fixture indeed acknowledges the control panel's beacon (step 1) by shipping an executable model to compute its expected energy consumption (step 2). The control panel locally runs the model (step 3) to compute an estimate of the light fixtures' energy consumption (step 4) before determining and transmitting its schedule (step 5).

SCENARIO C: if some devices run different platforms, the necessary coordination must rely on standard-compliant interfaces and inter-operable message formats. Such interfaces may also need to evolve after the system is installed, especially for **F3**. For example, landlords may decide to install solar panels and to sell the excess energy back to the grid. As shown in Figure 11, whenever this happens, the control panel should be extended with an additional interface to query the amount of produced energy. This interface will be used by the energy provider in the energy market. This is implemented by letting the newly installed solar panel answering the control panel's beacon (step 1) by requesting the addition of a new software component (step 2). This component will receive messages from the solar panel to periodically inform the control panel

about the produced energy (step 3). The same component will make this information available over the Internet, e.g., to the energy provider (step 4 and 5), in a standard and vendor independent manner.

```
Data: billing, requests, appliance models
Result: schedule for each appliance
while not solved do
    get best time slot from billing;
    foreach request do
        calculate appliance behavior;
    end
    foreach hour in time slot do
        if hour does not conflict with requests then
            fill with appliances until upper limit;
        end
    end
    if appliances is empty then
        solved = true;
    else
        calculate next best time slot;
    end
end
```

Algorithm 1: Schedule calculation

4

THE ELIOT FRAMEWORK

This chapter describes the ELIoT framework in detail: it shows the differences in syntax and semantics with respect to Erlang; it introduces the ELIoT Virtual Machine, a modification of the original VM, and we explain how the new language semantics have been implemented; then we describe the hardware interfaces, used to interact with sensors and actuators, and the hardware platforms where we ported the VM (Chapter 6 will show several performance measurements); we also show the dynamic RESTful interfaces that we introduced in ELIoT, based on the Computational REST framework presented before.

Running example

To make our explanation concrete, we consider the smart-home application introduced in the previous chapter. The ELIoT code in Figure 12 reports part of the implementation of the core functionality of the control panel: discovery of home appliances, as per functionality **F1** in the application base design (lines 20 to 29); gathering of the appliances' operating parameters, as per scenario **A** (lines 35 to 42); and installation of the executable model of an appliance's expected energy consumption, as per scenario **B** (lines 46 to 54).

The main structure of the example is as follows: after defining constants and structured types, the code in Figure 12 defines a recursive function **`receiver`** run by the control panel (line 15). It takes the current set of known appliances as input and assigns it to the **`Appliances`** variable. Processing stops at the **`receive`** statement (line 17) and then unfolds depending on the type of received message.

```
% Define some constants holding chars (1 byte) to be used as headers of messages
-define(BCON, $M).
-define(APPLIANCE, $A).
-define(APPLIANCE_LOCAL, $L).
% The timer for sending beacons
-define(TIMER, 60000).

% Define the 'appliance' record (tuple with named variables) with three fields: the
% appliance's IP address, the process id of the appliance's model (if running locally),
% and the list of its parameters.
-record(appliance, {ip, pid = none, parameters = []}).

% Main (recursive) function handling incoming messages. It takes a dictionary (key, value
% pairs) as a parameter, to hold the set of known appliances
receiver(Appliances) ->
  % Extract the first message from the incoming queue (blocking)
  receive
    [...]
    % On receiving the timer self message, build the beacon and send it in broadcast
    timer ->
    % Build the beacon with a single byte (8 bits): the value of constant BCON defined above
      Msg = <<?BCON:8>>,
      % Send the beacon, unreliably, to the processes called 'appliance' running on nodes
      % reachable from this one
      {appliance, all} ! Msg,
      % Re-send the timer self-message to myself, after TIMER milliseconds
      erlang:send_after(?TIMER, self(), timer),
      % Tail recursion: parse next message
      receiver(Appliances);
    % Process message coming from neighbors
    {RSSI, SourceAddress, Content} ->
    % Pattern match on the message content
      case Content of
        % First byte equals APPLIANCE, next is a binary blob: de-serialize and process
        <<?APPLIANCE:8, SerializedParameters/binary>>  ->
          Parameters = data:decode_params(SerializedParameters),
          % Create a new record with this appliance data
          NewRecord = #appliance{ip = SourceAddress, parameters = Parameters},
          % Add it to the dictionary (remember: immutable variables)
          NewApps = dict:store(SourceAddress, NewRecord, Appliances),
          % Tail recursion with the new set of appliances
          receiver(NewApps);
        % First byte equals APPLIANCE_LOCAL, next 20 bytes is a hash, then the lenght
        % (1 byte) of the following field (SerializedName), then a binary blob holding
        % serialized code: de-serialize and process
        <<?APPLIANCE_LOCAL:8, Hash:20/binary, L1:8, SerializedName:L1/binary, Code/binary>> ->
          Name = erlang:binary_to_list(SerializedName),
          % Spawn a new process to execute the given code (which is checked against the hash)
         {Pid, Parameters} = supervisor:start_model(Name, Code, Hash),
         % Create A new record for the appliance and add it to the dictionary
         NewRecord = #appliance{ip = SourceAddress, pid = Pid, parameters = Parameters},
         NewApps = dict:store(SourceAddress, NewRecord, Appliances),
         % Tail recursion with the new set of appliances
         receiver(NewApps)
      end
end.
```

Figure 12: Excerpt of control panel code.

4.1 THE ELIOT LANGUAGE

Here we describe ELIoT's dedicated language constructs, which concern three key aspects of inter-process communication when developing IoT applications:

- handling different communication guarantees
- supporting extended addressing schemes
- providing access to low-level information from the networking stack.

Interprocess communication in ELIoT

As mentioned in Section 2.1, Erlang inter-process communication is based on the `!` operator, which is equally used for sending messages to a local or to a remote process. In blurring the distinction between local and remote communication, Erlang assumes that the underlying protocol for sending messages among Erlang VMs is reliable, in particular the TCP protocol provides these guarantees to the developer. This is a strong assumption in the IoT scenarios we target, where wireless communication is the rule more than the exception. Indeed, TCP provides reliable channels under several key assumptions regarding the failure model, which is quite different on a wired network with respect to a wireless network, especially where a low power wireless protocol (such as ZigBee) is in place: channels may disappear due to nodes going out of range, the network is decentralized and properties like *broadcast transmission* have a different semantic (e.g., in a wired network they typically span the entire LAN, while in a ZigBee network they span the nodes that are visible to the radio of the sender in a specific moment, if there is no retransmission in the network protocol implemented in the application). At the same time, several IoT applications do not need reliable communication and may sacrifice that for better efficiency. Accordingly, ELIoT complements Erlang's `!` operator, with a new operator: `~`, which models unreliable, best effort, sending of messages. We see it at work at line 25 of Figure 12: after creating the

single byte beacon (line 22), the control panel sends it unreliably using the ~ operator.

```
1  % Extract the first message from the queue (blocking)
2  receive
3      % On receiving the timer self message, build a new message and send it
4      timer ->
5          Message = {Some, Content, Or, Another},
6          % Send Message to the process called 'destination' on a device named 'node1'
7          % at address 1.2.3.4, using reliable send
8          {destination, 'node1@1.2.3.4'} ! Message;
9      % If something very bad happens, I will receive this NACK...
10     {nack, ReceiverAddress, Message} ->
11         % ... and will react, e.g., by informing the user
12         notify_user("Sending to ~p failed", ReceiverAddress);
13 end.
```

Figure 13: Failure handling triggered by a failed message send.

Besides adding the ~ operator, ELIoT addresses possible faults of the underlying communication protocol by slightly changing the behavior of the ! operator: as pointed out before, the network cannot guarantee some properties anymore, and (as a consequence) neither the network protocol (details on the new protocol implementation will follow); this means that we needed to give the developer a way to know if the communication fails: in Erlang the *send* primitive returns immediately, regardless of the destiny of the message; in ELIoT, instead, in presence of communication faults that cannot be resolved, the framework places a special **nack** message into the sender's incoming message queue. Programmers can realize application-specific failure-handling mechanisms based on such notifications, as exemplified in Figure 13. When a timeout expires, the process prepares and reliably sends a message to a specific destination **Node1** (lines 4 to 8). The clause at line 10 matches the **nack** message that the underlying VM generates for the sender process, should the sending fail. In this example, the process simply reacts by notifying the user (line 12), yet programmers are free to implement smarter mechanisms to handle such situations, possibly based on the actual destination and payload of the failing massage, which are returned as part of the **nack** message.

More generally, the need to carefully control the costs associated with wireless communication—both in terms of energy and bandwidth consumed—hardly match the level of abstraction inherent in Erlang's original inter-process communication model. Explicitly providing a best-effort message send operator, alongside a more reliable one, reconciles the need for keeping a reasonably high level of abstraction with the reality of unreliable wireless communications.

Notice that ELIoT retains the blurred distinction between local and remote communication by allowing both message send operators to be used to communicate with local processes. In this case, both straightforwardly guarantee delivery of messages.

New addressing schemes in ELIoT

Through the `!` operator, Erlang provides solely unicast messaging. Single processes can be easily reached, being them local or remote, once programmers know their unique identifier or the name they registered to, together with the address of the VM they run on. While this is enough for the typical client-server communication Erlang was conceived for, it is not enough to efficiently support scenarios when a process needs to send a message to all other reachable processes. This form of broadcast communication is often used in IoT applications, either as a primitive at the application level, e.g., for discovery, or as a low-level mechanism to implement higher-level communication protocols. Even though Erlang libraries provide broadcast functions, they use multiple unicast communications to emulate real broadcast with an unacceptable waste of resources.

ELIoT, instead, supports these scenarios by offering a richer addressing scheme than Erlang. In particular, ELIoT messages addressed to `{n, all}` reach processes with name `n` running on all reachable VMs; the notion of reachability is a function of the target network scenario:

- the ELIoT version used for implementing the smart home scenario uses UDP as communication protocol and broadcast UDP for the `all` keyword, thus the span of message spreading depends on the network

configuration (typically, it works only in the LAN and it does not go across routers; it could be easily ported to IP multicast to overcome this problem);

- an ELIoT prototype tested over ZigBee networks uses the protocol broadcast addressing, thus the span of message spreading is limited to the devices in range.

Figure 12 (line 25) uses the **`all`** keyword to implement discovery of new appliances in the house network.

Notice that mapping processes to names happens in two steps. First, a process **`register`**s itself under a symbolic name, as in standard Erlang. This allows communication to the registered process based on its name, without knowledge of the process id that the VM assigns at run-time. The process then becomes accessible from the network only if explicitly **`export`**ed, using an ELIoT-specific function. Separating the two steps spares memory and processing overhead at the VM level for processes that do not require network interactions.

Addressing based on the **`all`** keyword has wide applicability in ELIoT. In particular, programmers may also use it with the **`spawn`** primitive. This is required, for example, when a new functionality is to be deployed on all reachable nodes at once.

To further control the individual nodes where such spawning must happen, programmers may use ad-hoc *scoping filters*. They express a condition—in the form of a lambda function—that predicates over environment variables the application supports or that invokes functions available within the application itself. The process is actually spawned only on those nodes where the scoping filter evaluates to true. We show an example of scoping filters, together with the ELIoT-specific **`spawn_cond`** primitive, in Figure 14.

Accessing low-level information from the ELIoT *networking stack*

Full isolation of the various layers that build a networking stack is sometimes impossible to achieve and often not beneficial. Indeed, some form of

```
1  % This function reads some temperatures from sensors and averages them
2  read_avg_temperature() ->
3      Values = read_temperatures(),
4      average(Values).
5
6  % This function checks if the 'temperature_sensor' variable is set in the ELIoT environment;
7  %  we expect it being defined only on devices actually equipped with temperature sensors
8  temperature_node() ->
9      % Check the environment
10     case application:get_env(temperature_sensor) of
11         % The variable is not set
12         undefined -> false;
13         % The variable is set (and we ignore the value of the variable itself)
14         {ok, _} -> true
15     end.
16
17 % Spawn function read_avg_temperature on all devices reachable from this one,
18 % but only on those equipped with temperature sensors
19 spawn_cond(all, read_avg_temperature, temperature_node).
```

Figure 14: Scoping filters.

cross-layering is often required to improve efficiency, especially in presence of embedded devices and wireless communication, which are the norm for IoT scenarios.

ELIoT makes these considerations concrete by exposing information coming from the network stack to the receiver. More specifically, while Erlang fills the incoming message queue of the receiver only with the payload of the message, hiding every other detail, the ELIoT *network driver* (part of the VM) explicitly exposes additional information. In the current prototype, the IP address of the source node and the Received Signal Strength Indicator (RSSI) coming from the radio are added, but the communication driver can be easily extended to add other information. Line 31 of Figure 12 shows how this information can be easily accessed. This sharply contrasts the way programmers access and process similar information using low-level embedded system languages, like C. Indeed, the IP source address and RSSI reading in ELIoT are treated as any other type of data, and automatically materialized by ELIoT into the receiver's incoming message queue, without requiring intricate platform-dependent code. As a result, ELIoT simplifies not only the development of application-level functionality, but also the implementation

of system-level services, e.g., RSSI-based localization algorithms [17] required for location-aware services.

4.2 THE ELIOT VIRTUAL MACHINE

Erlang was originally designed to run on embedded platforms: it started in the '80s as a language for telecommunication devices inside the Ericsson copmany. In time, however, it grew up to support a much wider range of scenarios, by means of a large set of libraries and a complex run-time infrastructure. For example, a standard Erlang distribution contains the VM and the base libraries, and the support for many different technologies, from NoSQL databases to CORBA middleware to queue systems and much more. The VM itself consumes large quantities of memory to load some of these modules and launch several services during startup.

To address this issue, we decided to develop a custom version of the VM for ELIoT, wiping off all the libraries that are not needed to run most programs[3]; we also changed several aspects of the VM mechanisms, in particular regarding the network stack.

The structure of an ELIoT deployment is shown in figure 15: the VM runs on the GNU/Linux operating system (and its variants, such as OpenWRT), with a network driver developed distinctly and adaptable to different network types (e.g., WiFi, ZigBee...). On top of the VM run the OTP libraries[4], the hardware interfaces to interact with sensors and actuators (depending on the device which is running ELIoT), and the ELIoT API (shown in Table 2), which contains a small number of functions that substitute the equivalent ones from Erlang standard library and allow the simulator to run the applications unmodified (cfr. Section 5.1). On top of these libraries runs the ELIoT application itself (CREST is a library offering the capability of dynamic RESTful interfacess, Section 4.4 will describe it in more detail).

3 These libraries can be re-added if necessary, but the network communication modifications may require them to be modified: only those provided by ELIoT have already been adapted to the new VM.

4 See footnote 3.

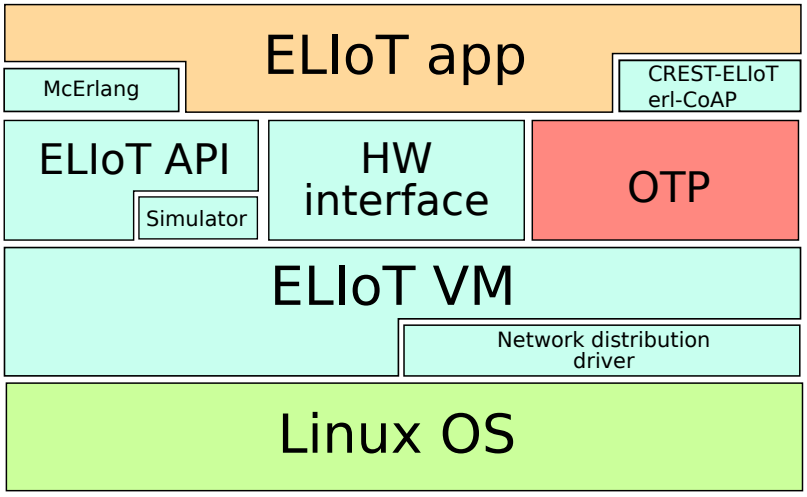


Figure 15: ELIoT structure

Table 2: ELIoT API.

Function	Description	Erlang equivalent(s)
`set_node_name`	Set the name of this ELIoT node	`net_kernel:start/1`, ...
`get_node_name/0`	Get the name of this ELIoT node	`erlang:node/0`, ...
`put_data/2`	Save global data as key-value pairs	`application:set_env/3`, `erlang:put/2`
`get_data/2`	Get global data	`application:get_env/2`, `erlang:get/1`

4.2.1 *The network stack*

The main modification to the Erlang VM regards the network stack that is used to allow the communication between processes located on different devices in a transparent way with respect to the developer; the following two sections will compare the existing approach and the ELIoT modifications.

Erlang

In the original VM, a process can obtain the identifier of another process regardless of the fact that it is run by the same VM instance, and exchange communication with it using the same **`send`** and **`receive`** calls that we pre-

sented earlier. The developer has to know the location of a process only if it is called by name (the node name has to be added in this case), and even in this situation a module called *global* is able to resolve names on the network and register them globally.

The standard communication protocol in Erlang, when traversing a network, has the following workflow (see Figure 16):

1. a process sends a message to a remote process (by name or by identifier)
2. the VM extracts the name of the remote node, and asks a local daemon (called *epmd*[5]) to resolve the name and contact the remote machine
3. the daemon contacts the remote machine daemon and checks if the node name exists
4. if so, the remote VM opens a new network port, and the two VMs now communicate directly with each other
5. a negotation phase (handshaking) takes place between the two VMs, to check if the two Erlang interpreter versions are compatible with each other (communication parameters can be set to make the dialog possible) and to check some basic security mechanisms (the *cookie*)
6. if the negotiation ended well, the two VM instances add each other to an internal list of known nodes, and start periodically pinging each other
7. finally, the initial message is delivered to the receiving process (if it exists, otherwise the message is discarded without alerting the sender).

The mechanism described at step 6 is important in Erlang, and it is needed to implement the Erlang version of broadcast: all the known nodes at a certain point in time are contacted through multiple unicast calls; if a ping fails to reach a node, then the node is removed from the list.

All the network communication runs over TCP/IP, which gives the guarantees on the reliability of the communication itself; a process has to implement

5 Which is actually a port mapper.

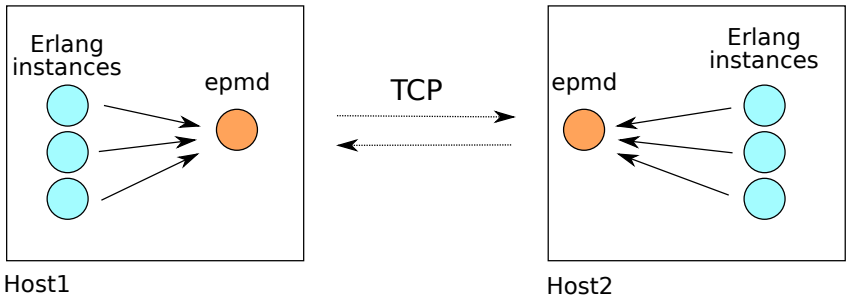


Figure 16: Erlang host intercommunication

a mechanism of acknowledgements if it really needs to know if a message has been correctly delivered[6], but by using this network protocol it knows that the network will try hard to make the message arrive at its destination.

ELIoT*: the driver*

In IoT scenarios, we decided that the developer needs more control over the network communication: in particular, she may not need reliability guarantees in some cases, and TCP/IP could be too demanding as a protocol when using, for example, different wireless network technologies. Given these reasons, we decided to drastically change the network communication.

The Erlang developers already provided the possibility of changing what they call the *distribution carrier*: it is a driver with a standard interface offering functions to start a new remote communication, send and receive serialized data (which is (de)serialized by the VM), and produce some statistics used internally to maintaing the known hosts list; the starting point has been to create a new driver using UDP instead of TCP as its network protocol; as a consequence, the communication steps have changed quite a bit (see Figure 17):

1. a process sends a message to a remote process (by name or by identifier)

6 Erlang has a function that can be used to *ping* explicitly another node, if needed.

2. the VM extracts the name of the remote node, and sends this information directly to the network driver (along with the message)
3. the network driver contacts the remote host on a standard UDP port, and immediately delivers the message: there is no negotiation anymore, because ELIoT uses a specific version of the communication protocol and parameters, and they are not negotiable
4. the receiving network driver takes the message and sends it to its local VM, which delivers it to the process (if it exists, otherwise the message is discarded without alerting the sender), and it handles the ACK mechanism described below.

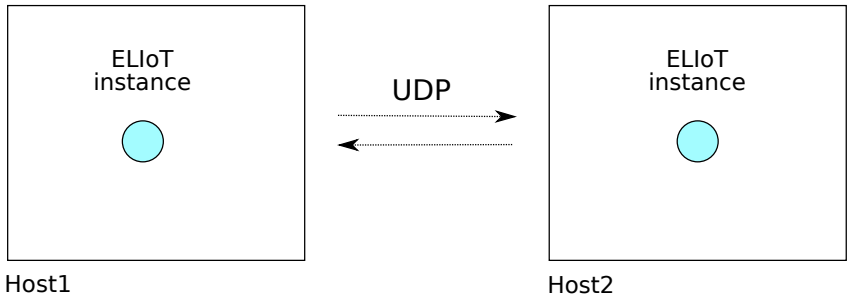


Figure 17: ELIoT host intercommunication

A distribution carrier driver is written in both C and Erlang: the C part handles the network communication, while the Erlang part takes care mainly of acting as a bridge between the C interface and the modules handling handshaking, pinging and creation of new instances of the driver, whenever a process sends its first message on the network.

The TCP driver worked in a way that each process that needed to communicate with the world created an instance of the driver, and each instance was associated to a specific TCP port; this means that there was a one-to-one correspondence between a process, a driver instance (called a *port* in the VM) and a network port: the standard Erlang network port 4369/TCP is used by the port mapper only for the initial communication steps.

By shifting to UDP, this correspondence lost its original meaning: UDP is stateless, so we decided to completely remove the port mapper[7]; now a driver instance listens on port 4369/UDP, and all messages are received by it, and it will deliver each message to the correct process; each process still has a correspondent instance of the driver, which is used only for sending messages. This mechanism made the driver a little more complicated than expected, to correctly handle the reception of a message for a process that never received anything up to that moment.

The new driver is also capable of real broadcast communication, and to implement this we used a simple trick: in ELIoT there can be only one instance of the VM per host, because of the use of a unique network port as described above, and the name of this instance (the name of the *node*, in Erlang terms) is decided *a priori* to be **`eliot@1.2.3.4`**, where the part following the *at* symbol is the IP address of the host; at this point, we arbitrarily chose **`all`** to be a way of indicating to the network driver that the communication should be sent in broadcast: **`all`** is a name of a fake ELIoT node, which corresponds to a specific instance of the network driver, created when the interpreter is started, that will send messages in broadcast. The devices that will receive the communication depend on how the network driver implements the broadcast communication: for example, our UDP driver implements it by using IP broadcast, while different implementations may use other mechanisms, depending on how their network protocol works.

Substituting the UDP driver does not break compatibility between ELIoT and Erlang *per se*, but as you can see from the communication steps presented before, we decided to remove the handshaking/negotiation part, and consequently the concept of overlay network that Erlang creates between its nodes: as with reliability, we decided that this is yet another part that can and should be implemented by developers only when needed in their scenarios, and the

7 Having an external process is also bad in some contexts, for example to run the framework on OS limited platforms such as Android, because it requires specific permissions that may be needed to perform some operation, requiring the developer to gain full control of the operating system of the device, an operation that usually voids the warranty.

VM should give them this possibility. Thus this makes the VM **not compatible** with the Erlang one.

ELIoT: *the send primitive*

As we described earlier, by removing TCP we removed the reliability mechanisms that were intrinsic in the network protocol: this means that a developer does not have any reassurance about its messages being correctly deliver, and moreover the Erlang primitives do not inform her about failures.

Given these premises, we decided to introduce a new operator, `~`, as described before, and change the semantics of the existing one, `!`. The `~` operator has the same syntax of the standard Erlang *send* operator, and locally it behaves in the exact same way; when sending a message to a remote process, however, the network driver simply encapsulates the message in a UDP datagram and sends it through the network, without any effort to check whether it has been received or not. A developer can ask the VM to try a little harder, by using the `!` operator: the network driver will request the receiver for an explicit acknowledgement, that will be handled by the driver—the VM will have no role in this. The driver will re-send the packet several times, and if no ACK has been received, it will inform the process sending the message, through a message delivered to its mailbox: the message will be in the form `{nack, Destination, Message}`, so the sender is able to distinguish the messages that it is sending out. At the moment, the acknowledgement parameters can be configured only in the network driver.

The implementation of this operator has been done by adding the operator itself to the language grammar, and by prepending additional information about the reliability to the serialized message that it is passed to the driver, thus making the VM (again) **not compatible** with the standard TCP driver. Figure 18 shows the path of a message from the Erlang code through several modules of the VM and back: both the BIF module (handling the *built-in function* calls) and the distribution module were involved in these modifications.

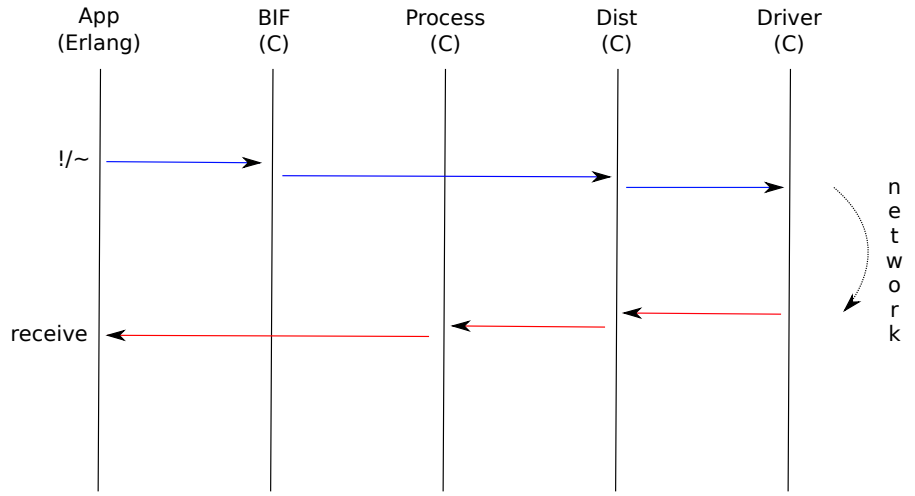


Figure 18: Path of a **send** call in the VM

ELIoT: *the protocol*

The final step in network protocol modification has been to change the protocol itself, more specifically the messages exchanged by the nodes. This change has been introduced to lower the impact of Erlang serialization, especially when comparing ELIoT applications with analogous ones written e.g., in C (see Section 6.2 for more details).

The Erlang network stack adds several informations to the original message, because each layer adds its own header:

- the original message is serialized
- then the VM appends a header containing informations about the type of the message and the destination[8]
- then the network driver adds its own header.

8 There are some 15 different messages in Erlang; here we are interested in the *SEND* and *REG_SEND* ones, that send data to processes addressed by identifier or by name.

The Erlang protocol is able to save space by adding an *atom cache*: atoms are cached so that they are not re-sent every time they are used in the communication; while this is a nice feature, ELIoT uses a previous version of the protocol that does not implement this cache, and this is due to the fact that UDP communication does not have the one-to-one correspondence between processes and driver instances, and atom caches cannot be correctly maintained (the VM does not know if a node has already sent messages before a specific moment in time, so it cannot maintain a reliable cache).

Thus, we decided to change the protocol by removing some unused fields; as shown in Figure 19[9], the driver adds 2 bytes, used for acknowledgement handling, then we decided to remove the name of the originating process and the cookie: the former was not used and not seen by the receiver anyway (we add the source address as an additional information), while the latter was not used because the cookie was handled during handshaking, and it was used to establish the overlay network that ELIoT does not support anymore. Then the driver adds RSSI and source IP address to the original message that will be put inside the receiver's mailbox. This is the third and last modification of the VM that makes it **not compatible** with the Erlang one.

All these compatibility modifications have also the consequence that OTP libraries that make use of the network are not compatible with ELIoT, unless they are modified to handle the additional informations (and the possible loss of messages); part of the standard Erlang libraries has been ported to this structure, but developers should test the use of external functions carefully.

9 Grey fields are used in the image for alignment purposes only.

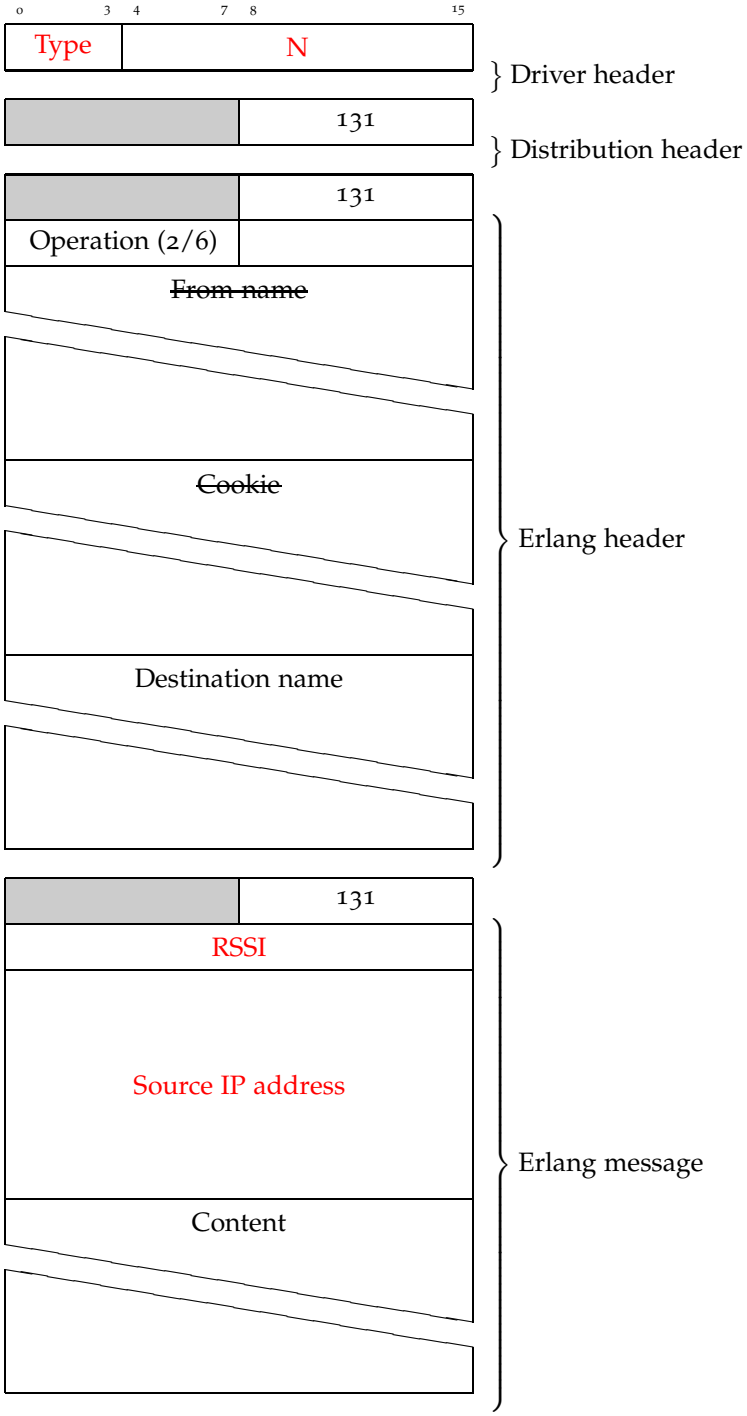


Figure 19: Network message in ELIoT

ELIoT*: other modifications*

The only other modification to the VM involves the ability of spawning processes on selected nodes: an additional parameter to the **`spawn`** call is a lambda function that will be evaluated on reception: if it evaluates to true, then the function will be spawned, otherwise it will be discarded. This mechanism, associated with the ability to spawn in broadcast (simply by using **`all`** as the node name), allows developers to select only parts of the network to be involved in the operation. An example usage could be to spawn a function only on those nodes that have some specific hardware features, e.g., specific sensors or actuators.

This does not break any compatibility, it simply introduces new functions in addition to the existing ones. The lambda function is quite limited in the current version: it has to be a one-liner containing a function call; the ability of sending more complex functions could be useful but the main problem is that Erlang needs to send the bytecode along with the lambda itself, otherwise the receiving end would not be able to execute it, and this involves more complex changes that are not supported by ELIoT yet.

4.2.2 *Supported platforms*

As a result of this work, our custom ELIoT VM drastically reduces the hardware requirements compared to Erlang's VMs, especially with respect to memory consumption. This enables ELIoT to run on devices that are quite unusual from those to which Erlang typically applies.

We tested different platforms (Table 3):

- a Raspberry Pi board (ARM processor), equipped with 256 MB of RAM and an external SD card
- an Android device (Nexus 7, ARM processor), equipped with 1 GB of RAM
- an ebook reader (Kindle 4th gen., ARM processor), equipped with 256 MB of RAM

Table 3: Hardware list

Device	Description	Capabilities	Limitations
Raspberry Pi	ARM device running Linux, 256/512 MB RAM, SD based, supports Ethernet and USB peripherals	Hardware interface: GPIO, SPI, I2C, UART; only one PWM output; several hw shields available	No ADC and only one PWM, means that hardware extensions are needed to interface with several hardware peripherals
Nexus 7	ARM multicore device running Android, 1 GB RAM, internal SD, supports WiFi	Internal sensors (accelerometer, GPS...), integrated touch-screen	Limited by the SL4A wrapper of Android API [18]
Carambola 1	MIPS device running OpenWRT, 32 MB RAM, flash based, supports WiFi	Hardware interface: GPIO, SPI, I2C, UART	Hardware interface has no PWM and quite low capabilities, especially for SPI and I2C; internal flash has only 8 MB of space available, which limits *a lot* what can be installed
Kindle 4	ARM device running Linux, 256 MB RAM, internal SD, supports WiFi	Almost none: it only provides an E-ink display; designed with power saving in mind	Slower than the other platforms due to soft-floating point implementation of the OS; it has no sensors or hardware interfaces, and the screen needs several libraries to be run. No SDK pubblicly available
Netduino 1	Atmel 32-bit microcontroller, KBs of RAM and disk	Hardware interface: GPIO, PWM, ADC, SPI, I2C, UART; compatible with several Arduino shields; it is a WSN-like device	It cannot run ELIoT, but it has enough capabilities to support a compatibility library with the network and (de)serialization protocols used by it
Waiting to be tested: Arduino Yún, Intel Galileo...			

- a custom embedded board with a RT3050 MIPS processor called "Carambola", featuring 32 MB of RAM and 8 MB of embedded flash

The latter currently represents the minimum hardware requirements to run ELIoT; the ebook reader has been tested mainly because of its power efficiency capabilities (see Section 6.2 to check the power consumption measurements).

We also developed a small library to show how fully-capable ELIoT nodes could interface with low-powered, WSN-like nodes and communicate with them, using a subset of the network protocol used by Erlang. In particular, we implemented this library in C#[10], and we run an example of a "low-powered appliance" able to communicate with ELIoT on a Netduino 1 [19]: this appliance is able to parse the beacon messages coming from the control panel and answer back with a binary blob containing a compiled Erlang module with the implementation of the appliance behavior; this module is then run internally by the control panel and it is used during the scheduling phase instead of communicating with the appliance itself (we suppose that the model implemented is quite complex and cannot be executed directly by the low-power Netduino board); this is an implementation of the scenario B presented in Section 3.2.

4.3 HARDWARE INTERFACES

```
1  readFromSensor() ->
2      % Open the SPI device communication
3      spidev:open("/dev/spidev0.0"),
4      % Send a request to the device and get the answer
5      case spidev:xfer2([16#68,16#0]) of
6          {ok, [A, B]} ->
7              % Combine the two bytes in a single value
8              Res = ((A band 3) bsl 8) bor B,
9              % Send the data to the network collector
10             eliot_ctp:collect(ctp, {eliot_api:get_node_name(), Res});
11         {error, Reason} ->
12             io:format(standard_error, "Error: ~p~n", [Reason])
13     end,
14     spidev:close().
```

Figure 20: SPI example: collecting data from sensors.

10 https://github.com/sivieri/eliotsharpmini

The scenarios ELIoT targets are typical IoT scenarios, where devices interact with sensors and actuators to obtain data and manipulate the environment; the devices briefly shown above either include some sensors (e.g., Android tablet has light sensor, accelerometer and gyroscope and many others), or include hardware interfaces to interact with custom boards: for example, the Carambola exposes several I/O pins supporting General Purpose I/O (GPIO), SPI and I²C protocols; using custom boards such as the one implemented for this project [20], the application would be able to obtain values from the sensors in the rooms and collect them.

For these reasons, ELIoT includes libraries supporting these interfaces: they provide a uniform API over different software implementations offered by the operative systems (e.g., OpenWRT running on Carambola and Linux running on the Raspberry Pi use different libraries to access the hardware, even if they both run the same kernel version).

The code shown in Figure 20 is an example of such behavior: the function interacts with a SPI device[11]: it sends two bytes (the first is the only one relevant to select the channel and the data format) and it receives two bytes back (SPI implements a full-duplex communication with each byte transfer); these two bytes are then combined to obtain the 10-Bit expected value. This value is then sent to the network device collecting data from all the producers, by using the Collection Tree Protocol [21] included in ELIoT.

```
1  getLocation() ->
2      % Begin GPS negotiation
3      android:startLocating(),
4      timer:sleep(15),
5      % Get the current location
6      Location = android:readLocation(),
7      android:stopLocating(),
8      doSomething(Location).
```

Figure 21: Get location on an Android device.

Android devices already integrate several sensors, and the library supporting Erlang and ELIoT on Android offers a layer to call Android API directly

11 In this example taken from a real-world application, a MicroChip MCP3002 Dual Channel 10-Bit A/D Converter.

from Erlang code [18], a higher level with respect to the hardware interface example shown before: as shown in Figure 21, an ELIoT application can access GPS coordinates and get the current location of the device, and then use that information to perform other calculations.

4.4 DYNAMIC RESTFUL INTERFACES IN ELIOT

This section introduces the integration of dynamic RESTful interfaces in ELIoT (from now on called CREST): we reused some of the concepts implemented in CREST-Erlang (discussed in detail in Section 2.2), which was a Web framework with the ability of dynamic reconfiguration of Web services, implemented with the Erlang programming language. This Web framework offers two main features to the developers, described in the next two subsections.

4.4.1 *REST interfaces*

The "Internet of Things" world is quite complex, as we described previously, with vendors offering devices with profound hardware differences and incompatible frameworks. One of the main jobs of the engineers is to combine different products to develop the solution required for the problem at hand. Because of this, providing an off-the-shelf RESTful interface to an IoT device becomes very important: it allows the use of a standard protocol to easily integrate different products.

This is the first functionality offered by CREST: an ELIoT node running the CREST framework introduces a new set of APIs to the developers (see Table 4), that can be used to install a module in the node (or list the running modules); the framework then takes care of wrapping this module with a REST interface, giving users the possibility of invoking it through HTTP operations. The module has to follow a simple structure to perform the wrapping; such structure is shown in Figure 22.

The entry point of the module should, at some point, start parsing inbox messages; in particular, there are three messages that should be supported

Table 4: CREST local API.

Function	Description
`crest:start_local/1`	Start a local process (in the form of a lambda function)
`crest:start_local/2`	Start a local process (module and entry point)
`crest:list_local/0`	List the installed modules

```
1  -module(example).
2  -export([example/0]).
3
4  example() ->
5      receive
6          {Pid, {"param", "name"}} ->
7              Pid ! {self(), ["Function example"]},
8              example();
9          % Specify the HTTP operation to be used to invoke this service
10         {Pid, {"param", "operation"}} ->
11             Pid ! {self(), ["POST"]},
12             example();
13         % Specify the number and type of parameters to be used when
14         % invoking this service
15         {Pid, {"param", "parameters"}} ->
16             Pid ! {self(), [{"samples", "integer()"}, {"interval", "integer()"}]},
17             example();
18         % Receive an invocation
19         {Pid, [{"samples", Samples}, {"interval", Interval}]} ->
20             % Send a message in broadcast using ELIoT
21             {temperature_sensor, all} ~ {sense, Samples, Interval},
22             % Receive the results
23             Results = receive_results(),
24             % Format the results as JSON and return them to the caller
25             Pid ! {self(), {"application/json", json:format(Results, ...)}},
26             example()
27     end.
```

Figure 22: CREST code example.

by the module, shown on lines 10, 15 and 19: through them, in a mechanism similar to reflection, the CREST framework discovers the HTTP operation to use[12] and the number and type of parameters that the module expects when receiving messages (line 19); from this information, the framework is able (through *reflection-like operations*), to create a REST interface that can be invoked by clients.

12 This may appear strange, given the fact that the framework automatically wraps the module in a Web service; if this line is not present, then the operation defaults to POST, but since GET and POST in HTTP have precise semantics [9], the developer should choose the most appropriate one.

The module can perform any kind of operations, remembering that each call like the one shown on line 19 could be a standard Erlang message from another process or an invocation coming from a Web client, thus being subject to HTTP timeouts if it takes too long to be completed. New Web standards (HTML5) introduced new concepts, such as Web sockets and asynchronous Web APIs, that can overcome timeout problems and allow clients to stream data continuously from a server (a very useful feature when showing data through charts, for example); ELIoT will introduce them in the next releases (see Chapter 8).

4.4.2 *Dynamic reconfiguration*

The second feature offered by CREST is a dynamic version of the API described before: a module can be installed on a remote node running the framework by invoking a specific URL, passing the bytecode of the module in a HTTP POST invocation. The framework also provides another Erlang API to obtain the same result (see Table 5).

Table 5: CREST remote API.

Function	Description	Erlang equivalent(s)
`crest:start_remote/3`	Install a function on a remote host	`http://www.example.com/crest/spawn`
`crest:list_remote/1`	List installed functions on a remote host	Web manager 25

The module to be installed can contain any type of code: for example, Figure 23 is an extremely simple function registering with a specific name and responding to "ping" requests.

If, however, the module is able to parse messages in the form of **`{name, value}`** lists, as shown in Figure 22, then it is automatically wrapped and exposed as a Web service, and the framework provides it with a REST interface that can be invoked remotely.

Finally, CREST introduces a local Web site to manage the installed services from a Web browser: in particular, the Web manager (Figure 25) lists the services on a specific host, allows developers to install a module directly from a

```
1  entry_point() ->
2      eliot:register(ping, self()),
3      eliot:export(ping),
4      example().
5
6  example() ->
7      receive
8          {RSSI, SourceAddress, {SenderName, ping}} ->
9              {SenderName, SourceAddress} ! pong,
10             example()
11     end.
```

Figure 23: CREST ping code.

browser (instead of using the previously shown URL or API), and automatically generates a Web form for each installed module that invokes the service wrapper, passing the input values for the given parameters (Figure 24 shows a very simple form for a service requiring a single parameter).

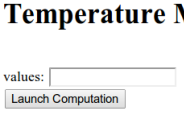


Figure 24: CREST Web form

4.5 CONSTRAINED APPLICATION PROTOCOL

Constrained Application Protocol (CoAP) is a software protocol that allows communication over the Internet for low-power devices, such as Wireless Sensor Networks, or devices that can be monitored and supervised remotely, such as switches and valves. It is currently published as a IETF draft, to become a RFC [22].

The protocol can be viewed as a simplified version of REST over HTTP: it is based on URIs and it implements a subset of HTTP operations (such as PUT, POST, GET and DELETE); it can be easily translated to HTTP for

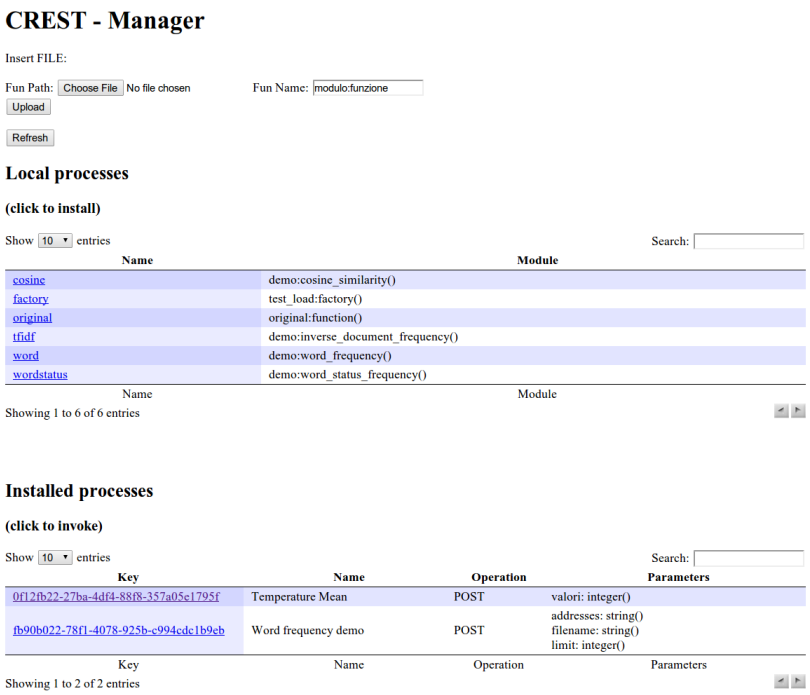


Figure 25: CREST Web manager

compatibility with standard Internet applications, but it also supports specific characteristics such as:

- multicast communication
- subscription to resources and push notifications
- low overhead.

It is usually implemented over UDP for devices supporting such protocols, or directly over IPv6 for devices such as WSNs.

As shown previously in ELIoT architecture, we included support for CoAP, too. Since CoAP is a protocol designed for IoT applications, it is important for our framework to allow compatibility with other IoT applications already

developed with different technologies. Thus, our library supports both client and server-side applications using this protocol.

The client library allows ELIoT applications to request resources to existing CoAP applications, and get the response. This allows, for example, a ELIoT application to interact with a Wireless Sensor Network, if the devices support IPv6 communication or a specific gateway, able to translate between different protocols, is in place. Our library has currently been tested over UDP and IPv4/IPv6, and we are currently working on a port to 802.15.4 networks.

The server library is used analogously to the dynamic RESTful interfaces: the framework automatically provides a CoAP interface for a module, and translates the request parameters to an Erlang message that can be received and parsed by the module, as it happens for REST requests.

For example, a module providing a sensor value can be reached both as `http://eliotserver/crest/UUID/temperature` and as `coap://eliotserver/UUID/temperature`, thus a ELIoT service can be queried also by low-powered devices, when needed.

As for the client library, the current version supports UDP and IPv4/IPv6 (depending on the device configuration), and we are currently working on a port to 802.15.4 networks.

5

THE ELIOT FRAMEWORK: DEVELOPER'S TOOLS

This chapter describes two important tools that the ELIoT framework and development platform offers to developers: a simulator, to test applications for algorithms scalability and devices functionality, and a model checker, based on [23], to prove the correctness of algorithms and applications written with ELIoT.

5.1 THE SIMULATOR

Debugging and testing distributed IoT applications is a key area scarcely supported by most programming platforms. Gaining the required visibility into the system state, in particular, is deemed to be an important feature for IoT development [24]. 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 different configurations.

Full simulation

A full simulation (Figure 27) is the execution of an application completely simulated: a single host executes a certain, configurable, number of ELIoT nodes, which will run on different hosts in the final deployment (Figure 26 as a reference).

In this configuration, the simulator uses its own version of the ELIoT APIs to abstract away the fact that different Erlang functions are actually used to simulate different nodes: in particular, process names are changed (e.g., a process registered as **`collector`** on a node **`node_1@10.0.0.3`** will be registered as **`collector_node_1`** instead), and this information is maintained in a process

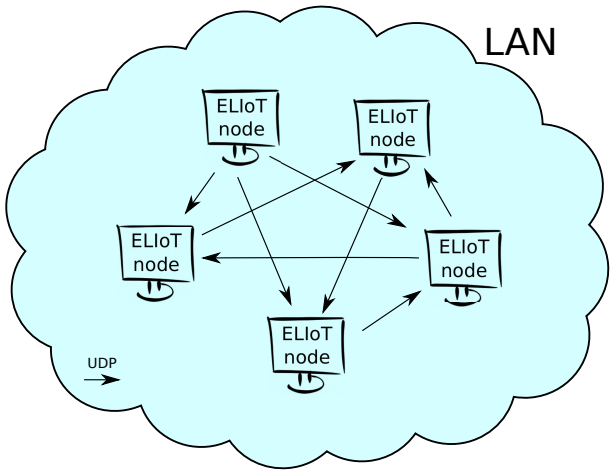


Figure 26: Deployment.

translation table, so that all the function calls referring to the process have to lookup in this table and correctly use the new name of the process.

This is done in a transparent way from the developer's point of view (see Figure 28): the application is compiled using a feature of Erlang called *parse transform*, which allows ELIoT to intercept the abstract syntax tree of the code to be compiled, and change it according to some rules specified in an ad-hoc module, all without having to create intermediate, semi-compiled files. This means that, for example, all the **`send`** calls are substituted with the simulator versions. The only difference from the developer's point of view is that she has to use a different compilation command (e.g., **`make simulation`** instead of **`make`**).

Table 6: Simulator API.

Function	Description
`register/2`	Register the name
`export/1`	Export the process on the network
`send/2`	Send a message (`!` and `~` too)
`spawn/1`, **`spawn/3`**	Spawn a new process

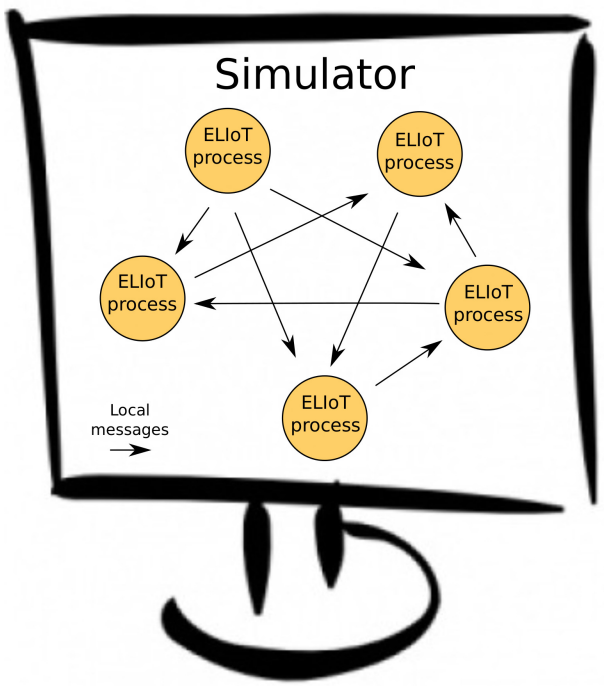


Figure 27: Simulated environment.

The full simulation capability can be used to test scalability of the algorithms in use: a deployment of a hundred nodes can be costly, while a simulation of the same number of nodes requires only a computer. The next configuration allowed by the simulator is to perform a *mixed* simulation (Figure 29): some nodes are simulated in a single host, while other nodes are real nodes installed on the final devices [13]. In this way, developers can test on the real hardware whether their code works or not (e.g., interfacing with hardware ports, whereas the simulator can only provide random values for their simulated equivalent), while at the same time the scalability of the application can still be under scrutiny.

13 The CREST framework cannot be simulated, it can to be tested only on real devices.

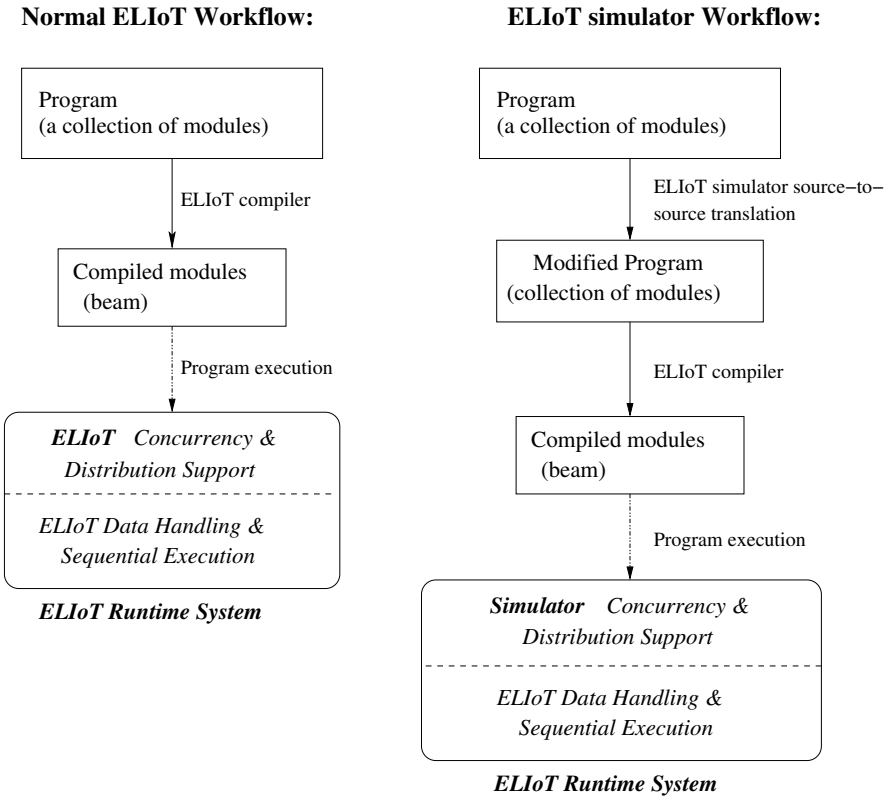


Figure 28: ELIoT simulator workflow.

Looking at the internals, the simulator starts additional processes that listen on the standard ELIoT port for external communication directed to any process; they check against the process translation table whether to route the message to the simulated node or not, and translate back if/when an answer has to be sent to the external network. The simulator does not make assumptions about the type of network on which it is run, as long as the network distribution driver supports that specific network (e.g., ZigBee vs. WiFi).

The simulation comprises the network links between each node: in particular, we use the TOSSIM network links modeler [25] to produce a set of traces that define parameters for the communication between each pair of simulated

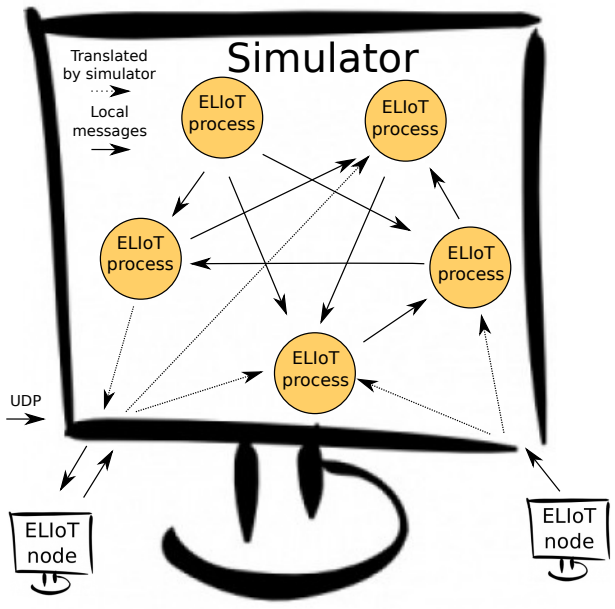


Figure 29: Mixed environment.

nodes; the simulated **send** operation leverages these traces to decide whether to deliver or not a message to the destination. If this probabilistic model is not enough for the purposes of the simulation, then a more complex model, for example maintaining a state for each channel, can be plugged in by implementing a simple Erlang interface.

Developers can access an interactive shell inside the simulator and inject messages in the network (to simulate sensor readings, for example, or to check the message hops in a collection algorithm), or monitor any debug message coming from any of the simulated nodes. They can also use the standard Erlang debugging tools to check the code: Figure 30 shows the simulator at work with the smart-home application. In this configuration, the control panel runs on a Raspberry Pi, while four appliances are simulated for debugging purposes. Developers interact with the ELIoT simulator in three ways: *i)* the process monitor in ⟨1⟩ shows the ELIoT processes running on

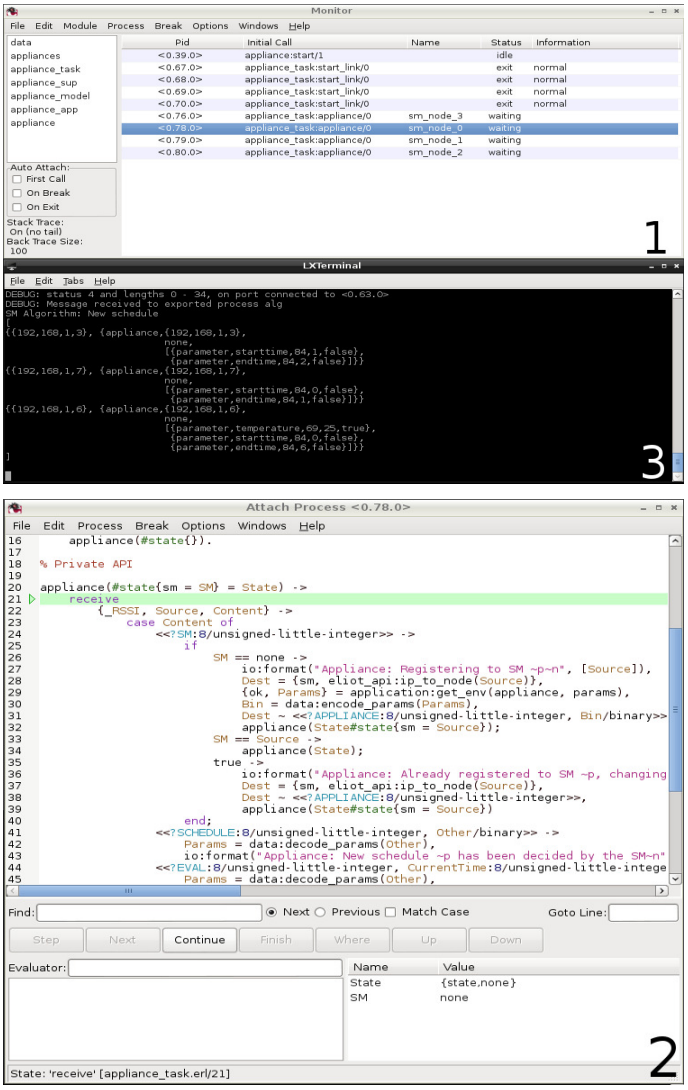


Figure 30: Erlang debugging tools using ELIoT simulator.

simulated nodes, identified according to their **`register`**-ed names; *ii)* selecting a process in ⟨1⟩ opens the code monitoring in ⟨2⟩ that enables inspection

of the currently running code—in Figure 30, the **appliance** process is blocked waiting for incoming messages—and allows to step through instructions and set breakpoints, as well as to inspect or to manipulate the values of variables; *iii)* the shell in ⟨3⟩ is bound to the real Raspberry Pi and allows developers to trigger specific executions, e.g., the computation of a new schedule for the appliances. When doing so, the simulator then shows how the appliances answer to the control panel through the process and code monitors. The shell allows automatizing these operations by scripting sequences of test cases.

To recap, the ELIoT simulator allows to start debugging a system in a fully simulated deployment, and then to 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.

The ELIoT simulator presents functionality to developers that are rarely available using mainstream programming platforms for networked embedded systems. The VM-based execution of ELIoT, together with the actor model that simplifies inter-process communications, facilitates building tools that effectively support developers in accurately testing and debugging distributed functionality.

5.2 MODEL CHECKER

Another interesting addition to the set of tools for ELIoT developers is a model checker, that allows them to write and test properties of their applications, with specific mechanisms to consider the unrealiability of communication channels.

5.2.1 *McErlang*

McErlang [23] is an existing model checker for Erlang applications, written in Erlang. The fact that the same language is used both for the analyzer and the

application helps the former to simulate in an effective way the peculiarities of the language.

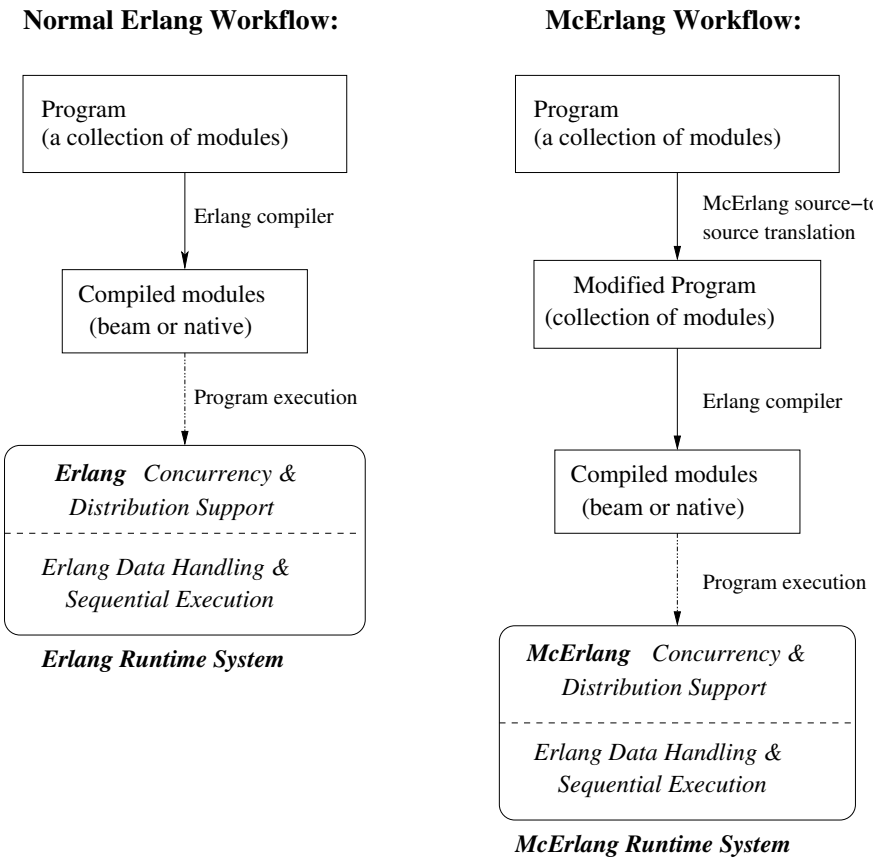


Figure 31: McErlang workflow (from the McErlang manual).

As described by the authors, the workflow of the verifier is somewhat similar to the workflow of our simulator (Figure 31): the source code is modified on the fly by *parse transform* mechanisms so that the *pure* parts of the application (and the garbage collection) are executed directly by the Erlang VM without changes, while the side effect parts are executed by McErlang, using rewritten parts of the standard library. In particular, the library functions re-

lated to process handling (creation, scheduling and communication) and fault tolerance have been substituted with Erlang native code (instead of part Erlang, part C code, as described in the previous chapters when discussing the VM internals). An application is executed in a single host, exactly as with our simulator, and not in a distributed fashion.

The model checker offers several different *monitors*, which are functions called whenever a new state is entered, and their job is to determine whether the state satisfies the condition expressed through that monitor. The default one is a *safety* monitor, which checks a safety property, e.g, if a deadlock may arise between the processes in execution.

However, the monitor that mostly interests our discussion here is the one implementing *Büchi* automata. This monitor is built starting from a *Linear Temporal Logic* property, which is translated to a *Büchi* automaton implemented in Erlang, which is then compared against the state tree explored during the application simulation; the monitor signals an error if it finds an infinite loop containing only accepting monitor states.

An example

The following example explains the basic functionality of the model checker, as described above. The code in Figure 32 simply generates a parallel process, which sends a message to the generating one, which it turn answers to it. When the second process receives the *echo* of the first message, it terminates its execution. Now, suppose we want to check the following property: *eventually P*, where the predicate *P* is the fact that the echo has been received. To be able to check this property, McErlang needs to know when this fact happens, and to do so we call the function in line 7, which registers a new *probe*, with a name and a value as its argument: it marks the reception of the echo message, and from that moment on the predicate *P* holds true.

At this point, McErlang needs a couple more helper functions that receive the list of probes for each state, and link that specific probe to that specific predicate (as you can image, you can have multiple predicates and multiple conditions). The checker has a library that translates the LTL property ex-

pressed before into an automaton, and then executes the code against that automaton. In the end, the result is positive: no counter-example has been found by the model checker, so the property is verified for that code.

```
1  start() ->
2    register(echo, self()),
3    spawn(fun() ->
4              echo ! {msg, self(), 'hello world'},
5              receive
6                {echo, Msg} ->
7                    mce_erl:probe(received, Msg),
8                    Msg
9              end
10         end),
11   receive
12     {msg, Client, Msg} ->
13       Client ! {echo, Msg}
14   end.
```

Figure 32: Correct code.

On the other hand, the code in Figure 33 behaves in a completely different way, even if on a superficial reading it may seem equivalent to the previous example. The problem here lies in the fact that the process receiving the first message is spawned in parallel with the process sending the message: we know that the Erlang **spawn** primitive returns when the process has been created, but the VM may decide to postpone its execution; this means that, in some configurations, the second process may send the message to a non-existing process, while in other configurations it may work as expected. But, since we are verifying a LTL property, the *Büchi* automaton finds a counter-example: in some cases the second process does not answer to the first one, because it was not ready when the message was sent, so it has not been received. This means that the property does not hold anymore, and the model checker is able to find a counter-example (and it can show the execution trace that brought it to that state).

The main limitation of the standard version of McErlang, from our point of view, is that it treats the **send** operation as deterministic: a message sent from a process to another will be surely received; in ELIoT, though, things are quite different: the message may not arrive, because there is no network layer

```
1  start() ->
2    spawn(fun() ->
3              register(echo, self()),
4              receive
5                {msg, Client, Msg} ->
6                    Client ! {echo, Msg}
7              end
8      end),
9    spawn(fun() ->
10             echo ! {msg, self(), 'hello world'},
11             receive
12               {echo, Msg} ->
13                   mce_erl:probe(received, Msg),
14                   Msg
15             end
16         end).
```

Figure 33: Error code.

trying its best to deliver it (especially when using the new send operator). This means that the code being tested should be aware of this limitation, and introduce reliability mechanisms when necessary. This in turn means that a developer may want to verify if these mechanisms work in its scenario, knowing (at least to some extent) the characteristics of the environment.

5.2.2 *McErlang in* ELIoT

To show the type of verification that a ELIoT developer may want from a model checking tool, take the code in Figure 34: it is an excerpt of the code used in the scenario (see Chapter 3) to calculate the optimal schedule for the appliances in the smart home. The function is fairly complicated, but the point here is what happens on line 18: the call to the **rpc** function sends a message to another ELIoT node, and it expects an immediate answer: this is due to the fact that the schedule being built needs consumption prediction by all the appliances in the house, several times during the calculation, to obtain the optimal time slots to engage each appliance.

The **rpc** function can be as simple as the one shown in the same figure, on line 43: it sends a message (using the more reliable **send** operator) and blocks until the answer has been received; if, for some reasons, the network layer is

not able to get through with the message (even with packet re-send), then the scheduling algorithm will get stuck and it will not be able to perform its task. The developer should find an alternative algorithm to avoid this problem, if she wants to guarantee the packet delivery, and she would want to be able to verify her solution and check if the application satisfies the requirements.

Modeling channels

With this goal in mind, we decided to modify the McErlang application and add a mechanism to model the channel packet reception rate (PRR), which depends on the environmental conditions of the deployment. Several parameters can be used to model the channel (e.g, the *beta* [26] and *gamma* [27] factors), but as a first step we imagine the developer knows the environment and she is able to describe the channel quality in terms of *conditional packet delivery functions* (CPDF) [28], which essentially describe the probability a packet will be delivered successfully after n consecutive failures or successes.

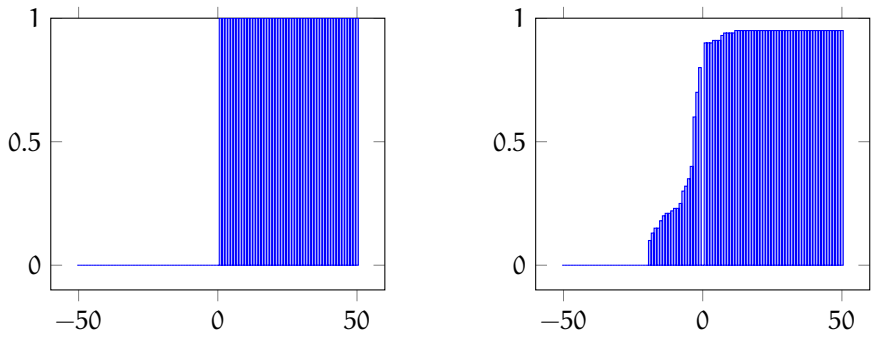
In practice, the channel is modeled as functioning in two states: sporadic and burst. In the first case, the probability of a packet to be delivered to the receiver is independent between each single transmission, because the time intercurring between each of them is such that the different transmissions do not influence each other's probability. When, however, the channel is under a burst of subsequent packet transmissions, the probability is no more independent, and the model expresses that by describing the probability of success (failure) after a certain number of successes (failures).

For example, Figure 35 shows two channel models: the X axis shows the current number of transmissions, where a positive value means success, and a negative number means failure (e.g., 5 means "5 subsequent successes", whereas -3 means "3 subsequent failures"). The first chart is the perfect (theoretical) situation, where packets are always delivered (positive numbers have a probability of 1), and no failures occur (negative numbers have probability 0). The second chart shows a more realistic situation, where, for example, after 2 transmissions, there is a 0.9 probability of another succes and a 0.1

```
calc_single_app(Cur, CurConsumption, Cap,
                #appliance{ip = IP, pid = Pid, params = Params} = Appliance) ->
    #parameter{name = starttime, value = Start} = hd(lists:filter(filter_1/1, Params)),
    #parameter{name = endtime, value = End} = hd(lists:filter(filter_2/1, Params)),
    RealCur = Cur rem 24,
    Dest = case Pid of
        none ->
            {sm, eliot_api:ip_to_node(IP)};
        _ ->
            Pid
    end,
    if
        RealCur >= Start andalso RealCur < End ->
            Bin1 = data:encode_params(Params),
            Message = <<?EVAL:8/unsigned-little-integer,
                        RealCur:8/unsigned-little-integer,
                        Bin1/binary>>,
            Res = case rpc(Dest, Message) of
                <<?EVAL:8/unsigned-little-integer, Consumption:16/unsigned-little-integer>>
                    when Consumption + CurConsumption =< Cap ->
                    {Appliance, Consumption};
                _Other ->
                    NewParams = lists:map(param_map/1, Params),
                    {Appliance#appliance{params = NewParams}, 0}
            end,
            Res;
        true ->
            {Appliance, 0}
    end.

filter_1(#parameter{name = starttime}) -> true;
filter_1(_Parameter) -> false.

filter_2(#parameter{name = endtime}) -> true;
filter_2(_Parameter) -> false.

param_map(#parameter{name = starttime, value = Value} = Param) ->
    Param#parameter{value = Value + 1 rem 24};
param_map(#parameter{name = endtime, value = Value} = Param) ->
    Param#parameter{value= Value + 1 rem 24};
param_map(Param) -> Param.

rpc(Dest, Message) ->
    Dest ! term_to_binary(Message),
    receive
        {_RSSI, _Source, Content} ->
            binary_to_term(Content)
    end.
```

Figure 34: Sample code to be verified.

probability of failure; at the same time, the probability of a third failure after two packets already failed is 0.3 (thus, 0.7 of success).



(a) Perfect PRR (b) Variable PRR

Figure 35: CPDFs

An important property of these models is that they limit the number of successive failures: after a certain number, the probability decreases towards zero (the shape of the probability depends on the channel properties). The charts consider a maximum value of 50 successful packets, after which there is no dependence between the first and last of the 50 samples.

These charts show a value of zero for the initial value of transmissions (when x equals to zero): this is due to the fact that the first send of a packet happens during a non-burst situation, and it may start a burst if the subsequent **send** operations happen immediately after the first transmission.

During the sporadic phase, we can describe the probability of success or failure of a single packet using the following formula:

$$p_n = 1 - (1 - p_1)^n$$

where p_n is the probability of success of the n-th transmission after $n - 1$ subsequent failures, and p_1 is the probability of success of a single transmission: the latter can be derived from packet delivery ratio (PDR) or known *a priori*. Thus, this formula generates the plot shown in Figure 36, for example for $p_1 = 0.5$. This mathematical function generates an asymptote for the

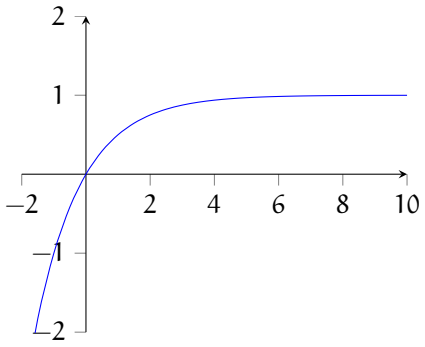


Figure 36: Non-burst plot with $p_1 = 0.5$

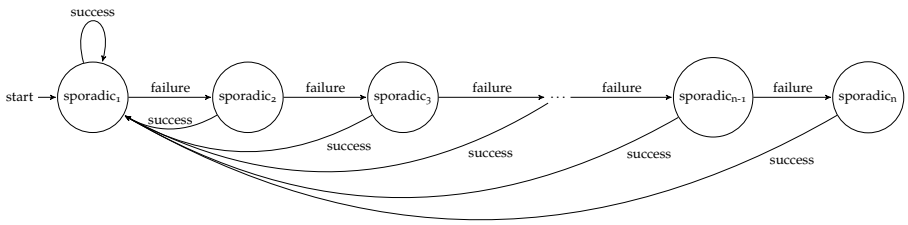


Figure 37: Non-reliable operator model.

probability on the y-axis equal to 1, but in our case we need to guarantee a maximum number of subsequent failures, otherwise the verification tool will generate a path that will never guarantee safety properties; this means that our mathematical function will reach the value 1 after a certain number of failed send operations.

McErlang modifications: theory

We need to distinguish how the tool considers the channels status, depending on the use of the reliable or non-reliable operator. In particular, non-reliable send operations (the ~ operator) behave as shown in Figure 37:

1. the tool follows the sporadic model shown in Figure 36, since the network layer in this case does not try to re-send the message if it fails; the

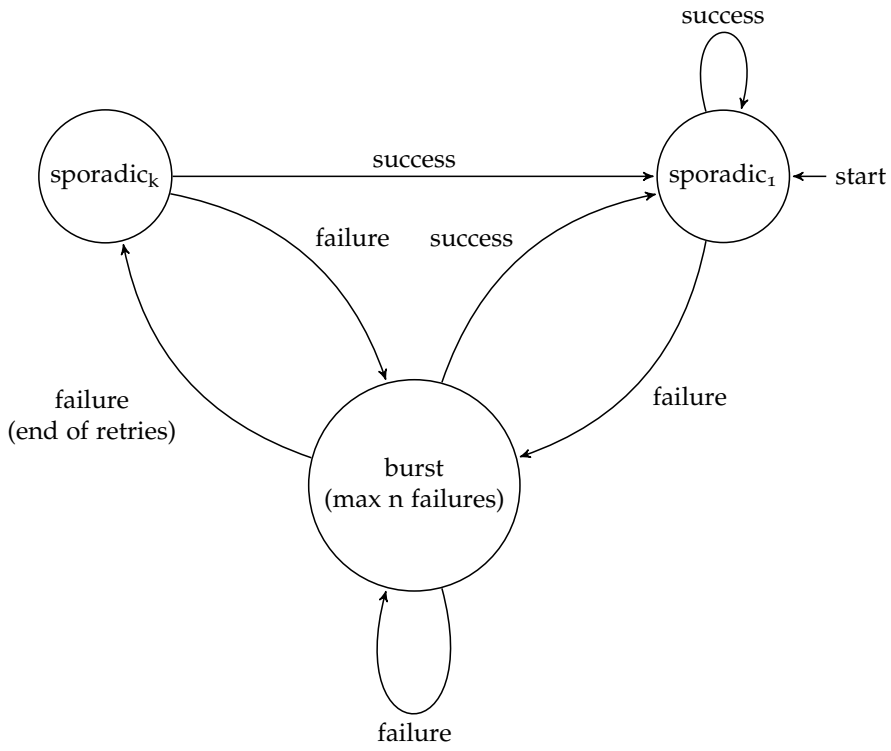


Figure 38: Reliable operator model.

first step is considering the probability p_1 for the first send (probability of the first send after zero failures): if it succeeds, then it will follow the same probability for the next send

2. if it fails, the tool will move to the next value the model, and the next send operation will behave as before, moving along the model if failures continue to happen

3. when the maximum number of subsequent failures has been reached (n in this case), then the send operation must succeed and the tool will go back to the initial state.

If the application uses the reliable operator (!), then the network layer will apply a certain number of re-send operations if the message does not reach the receiver; after a certain number of retries, it will give up and communicate the event to the application layer (NACK). In this case, we apply the burst model for the retries, and the sporadic model for the initial operation, as shown in Figure 38:

1. the model starts from the same initial step shown before, when the first send after zero failures is being performed; in case of success, the tool continues to follow the same probability value
2. in case of failure, it moves to the burst model and starts the retries: if it continues to fail, then it continues to follow the same model until one of the following two conditions apply:
   - the message is received, the tool goes back to the initial state and the next send operation will follow the initial value of the sporadic model, or
   - the network layer gives up, and the message is lost. In this case, the tool moves to the sporadic probability, but not at the initial value, rather at the k-th value (where k is the previous value of the counter in sporadic mode, before the burst, incremented by 1)
3. regardless of how the tool exited the burst model, the next send will move to the initial value of the sporadic model if it succeed, or it will move back to the burst state if it fails (and start a new batch of retries).

Moving from burst to sporadic

We performed a few tests to establish a value for variable k as shown on item 2 in the non-reliable model described before. The choices were either to perform an increment by 1, thus considering the burst section as a single transmission in sporadic mode, or to increment by the number of failed retransmissions during the burst phase.

We used the Twist Wireless Sensor Network testbed[14] such that each node was sending either a single broadcast transmission (95% of the time) or a 120-packet burst (5% of the time) in a 500ms time span. Each node logged each received message.

The experiment ran a total of 42 hours. We then took the logs of each node, and for each link we calculated several metrics considering both the two possibilities described eariler, and the mean error on PDR is 4.27% considering the burst as a single sporadic transmission, and 39.57% considering the burst as multiple retransmissions. Thus, in our implementation we are considering the increment by 1, since the error is one order of magnitude less than the second possibility.

McErlang modifications: implementation

We introduced this channel model by leveraging an existing McErlang function, called **`mce_erl:choice`**, which adds non-deterministic features to the model checker. In particular, the functions accepts a set of functions, and non-deterministically chooses one of them for the execution of the application; when using *Büchi* automata, the model checker explores all the ramifications given by the function set, considering all the possible alternatives.

In practice, each **`send`** generates two states, by using the **`choice`** function: one in which the transmission was successful, the other in which it failed; a global state of the channels for each pair of (simulated) nodes is maintained[15], so each new application state knows the previous channel state, and it updates the latter with the new number of subsequent successes or failures, and the current probability to reach that particular state.

As an example, Figure 39 shows a very simple channel model, that considers at most three failures one after the other; a **`send`** operation in this model would generate the execution tree shown in Figure 40 (which considers also the sporadic model seen before, with $p_1 = 0.5$). The tree on the left shows the probability of behaviors from the root to each leaf, where left branches

14 `http://www.twist.tu-berlin.de/wiki`.

15 This means that the models could be different for each pair, even if in the discussion here they are considered to be the same for the entire network, to simplify the examples.

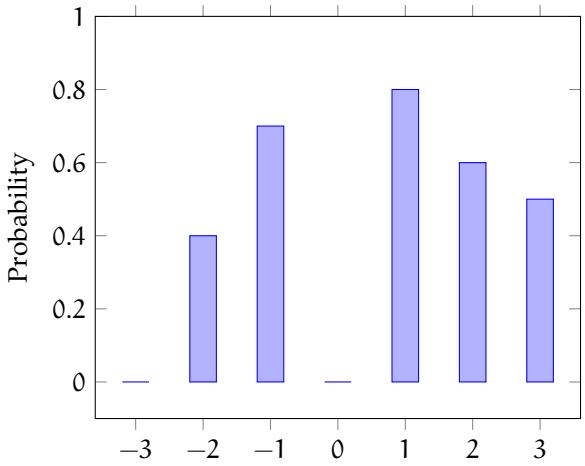
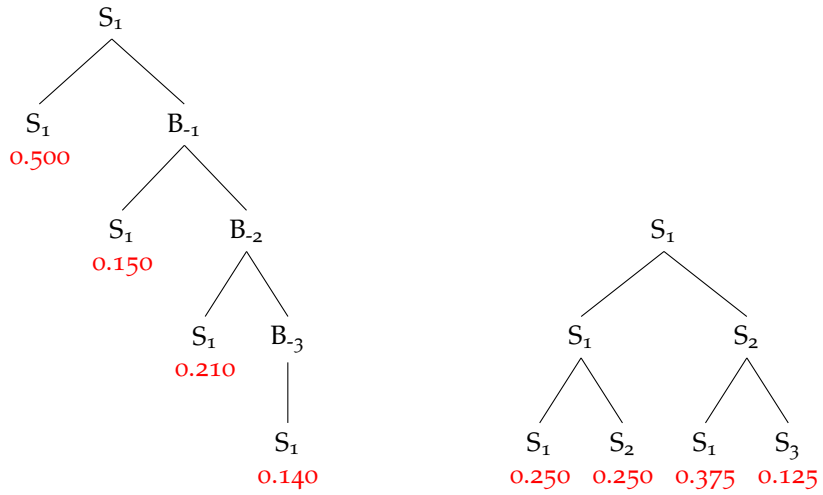


Figure 39: Simple channel model.

are successful transmissions and right branches are failures: as predicted, the first failure brings the tool to the burst model (the B nodes) when using a reliable operator, and it returns to the sporadic model (S nodes) whenever a successful transmission goes through. If the application uses a non-reliable send, as shown on the right, then it will use values only from the sporadic model, and the tree shows how two subsequent transmissions behave.

The output of this version of the model checker can be as follows:

- the property is not satisfied, and a counter-example is found; in this case, as with the standard version of McErlang, the developer can see the counter-example and the trace of the execution that brought to that specific error, and try to solve the issue;
- the property is satisfied, and the tool produces probabilities for each of the leaves of the execution tree; the developer can then take these values and analyze the behavior of her application: these data can tell, for example, the probability of having x number of retransmissions during the execution of the application, which can be useful to know the prob-



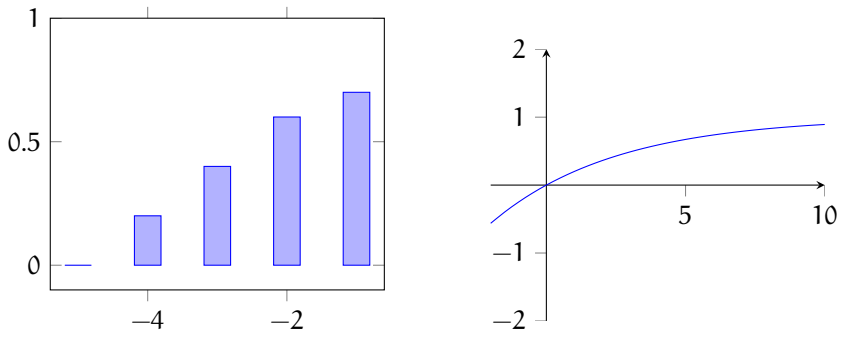
(a) Reliable send (b) Non-reliable send

Figure 40: Send trees

ability of having the radio active for a certain period (retransmissions require more energy).

As an example of a possible bug that the verification tool is able to detect in this way, let's look back at the code in Figure 34: each iteration of the algorithm sends a message from the control panel to an appliance, and watis back for an answer; both transmissions use the reliable operator, thus we can apply the two models that we saw earlier. For the sake of this example, let's assume that the burst model is slightly different from the previous one, in particular the number of subsequent failures is 5 instead of 3 (Figure 41, left, which reports only the negative part of the chart), but the framework is tuned to perform at most 3 retries of a failed send operation. The sporadic model shown on the right in Figure 41, for $p_1 = 0.2$ and reaching probability 1 when $x = 10$.

The tree resulting from a single exchange between the control panel and an appliance is shown in Figure 42, and the developer is verifying a property stating that *eventually the schedule will be computed*: this property does not hold



(a) 5-failure channel model

(b) Non-burst plot with $p_1 = 0.2$

Figure 41: Models for the smart home example

for all the leaves in the tree, since the retries are not enough to overcome the maximum number of failures during the burst. Thus, the tool is able to find the first of the tree paths that leads to a violation of the property, and the execution trace shown to the developer will show the exact sequence of function calls and send operations that bring the application to an inconsistent state.

At this point, the developer can decide, for example, to increment the number of retries, at the expenses of power and network traffic, or to change the algorithm so that it will contact the appliance again if it does not respond immediately. The original McErlang tool, lacking the channel models, would have returned no counter-examples, thus the application (once deployed) would have missed the capability of correctly computing the schedules in some of its iterations, possibly incrementing the current consumption of the smart home.

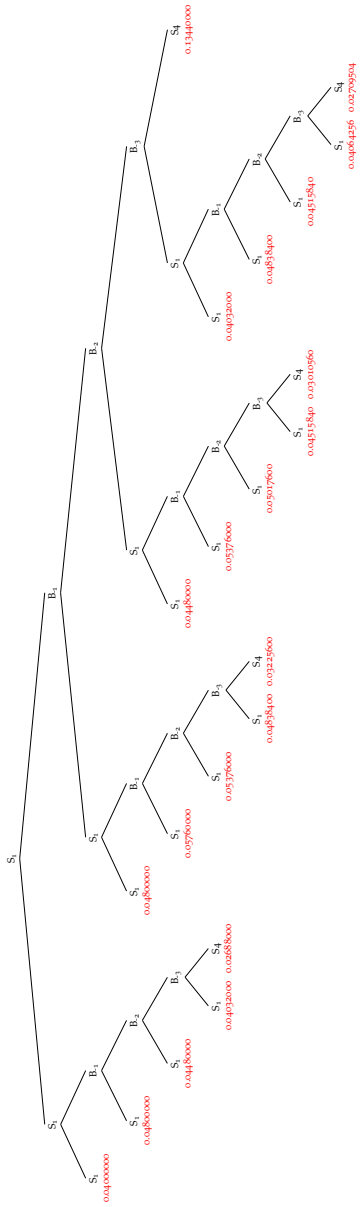


Figure 42: A - B exchange.

6

EVALUATION

This chapter describes the evaluation analysis that we performed first and foremost of ELIoT, primarily in an implementation of the home scenario described in Chapter 3, compared to a C implementation of the same application; this comparison is performed both qualitatively, i.e., looking at code readability and structure of the applications, and quantitatively, i.e., looking at memory and CPU consumption, network throughput and power consumption, on the various devices already introduced before.

Finally, it shows the performances of the REST interface implementation against the native functionality offered by Erlang to install new code at runtime.

6.1 QUALITATIVE EVALUATION

6.1.1 *Benefits to IoT Software Development*

ELIoT provides two benefits to programmers: it increases their productivity by rising the level of abstraction with respect to low-level languages like C, and it eases debugging with custom tools. These two aspects are separately analyzed next.

Programmers' productivity

It is notoriously difficult to objectively compare the implementation effort using different programming languages. Measuring the lines of code provides a rough, yet quantitative indication of such effort. In our case, the C-based smart home application requires 1623 lines of code, while the ELIoT-based implementation merely requires 649 lines, corresponding to a 60% saving.

Table 7: Home scenario application line code comparison.

C	ELIoT	ELIoT (manual code)
1623	649	462

Such figures of improvement become even more relevant as one considers that the C implementation only provides the core functionality of the smart-home application. Indeed, 187 lines of ELIoT code, out of the 649 total, are actually used to set up the OTP's application supervisor to provide failure handling against process crashes, and to configure testing and debugging services. These functionality are not available in the C implementation. Nevertheless, these fragments of ELIoT code are largely borrowed from existing templates; thus the number of application-specific lines of ELIoT code is effectively 462, for a 71.5% reduction with respect to the C implementation.

Beyond the raw numbers, using ELIoT caters for a higher level of abstraction that improves code readability, facilitating reuse and maintenance. This becomes visible by looking at the *structure* of the control panel code, shown in Figure 12. This structure is typical of ELIoT applications that implement communication protocols. The code is organized as a single **`receive`** statement with multiple cases, each associated to a specific message type determined in a declarative fashion by pattern matching.

As an example, line 2 in Figure 43 uses binary pattern matching to determine when the message payload contains a function to be executed locally. Matching happens in blocks: the first 8 bits are interpreted as a user-defined code indicating the message type; the next 20 bytes are a SHA-1 hash code; then a single byte specifies the length of the string that follows. Variable **`L1`** is assigned the latter value and immediately used as the length of the next field, namely the function name. The rest of the sequence is a binary block that holds the function's byte-code[16]. The name, hash, and code of the received function are then passed to the application supervisor (line 49) to spawn a new process executing the code and to monitor its execution for reacting should run-time errors occur.

16 The bit syntax allows to specify the length of each field using different units (bits or bytes), depending on the field's type.

```
1  [...]
2  receiver(Appliances) ->
3    receive
4      [...]
5      timer ->
6      % Build the beacon with a single byte (8 bits): the value of constant BCON defined above
7        Msg = <<?BCON:8>>,
8        % Send the beacon, unreliably, to the processes called 'appliance' running on nodes
9        % reachable from this one
10       {appliance, all} ! Msg,
11       [...]
12       case Content of
13         % First byte equals APPLIANCE, next is a binary blob: de-serialize and process
14         <<?APPLIANCE:8, SerializedParameters/binary>>  ->
15           Parameters = data:decode_params(SerializedParameters),
16           [...]
17           % Tail recursion with the new set of appliances
18           receiver(NewApps);
19         % First byte equals APPLIANCE_LOCAL, next 20 bytes is a hash, then the lenght
20         % (1 byte) of the following field (SerializedName), then a binary blob holding
21         % serialized code: de-serialize and process
22         <<?APPLIANCE_LOCAL:8, Hash:20/binary, L1:8, SerializedName:L1/binary, Code/binary>> ->
23          [...]
24          % Tail recursion with the new set of appliances
25          receiver(NewApps)
26       end
27   end.
```

Figure 43: Excerpt of code from Figure 12
(shown here for reader's convenience).

Figures 44 and 45 provide additional insights into the expressive power of ELIoT. In particular, they focus on deserializing the operating parameters of a newly discovered appliance (see line 15 of Figure 43). In C, as shown in Figure 44, this requires writing error-prone code that explicitly manages type conversions, memory allocation, and copying. Developers achieve the same functionality recursively and in a declarative fashion with ELIoT, using the binary pattern matching operators, as illustrated in Figure 45. In particular, the **`decode_params`** function in line 3 of Figure 45 takes the message payload as input and invokes a function with the same name and an additional argument: an initially-empty list of appliance's operating parameters. In line 7, if the payload is empty, indicating that message deserialization is complete, the list of deserialized parameters is returned as the final result. Otherwise, the first parameter is matched and decoded (lines 11 and 12). Each parameter includes the length of the parameter's name (**`L1`**) followed by the name

```
1  int deserialize_params(char *buf, GList **params) {
2    unsigned int params_len;
3    int tot, i;
4    parameter_t *param = NULL;
5    memcpy(&params_len, buf, sizeof(unsigned int));
6    for (i = 0, tot = 0; i < params_len; ++i) {
7      tot += deserialize_parameter(buf + sizeof(unsigned int) + tot, &param);
8      *params = g_list_append(*params, (void *) param);
9    }
10   return sizeof(unsigned int) + tot;
11 }
12 int deserialize_parameter(char *buf,
13                           parameter_t **param) {
14   unsigned long name_len;
15   parameter_t *p = NULL;
16   p = malloc(sizeof(parameter_t));
17   memset(p, 0, sizeof(parameter_t));
18   memcpy(&name_len, buf, sizeof(unsigned long));
19   p->name = g_string_new_len(buf + sizeof(unsigned long), name_len);
20   memcpy(&p->type, buf + sizeof(unsigned long) +  name_len, 1);
21   memcpy(&p->value, buf + sizeof(unsigned long) + name_len + 1, sizeof(uint8_t));
22   memcpy(&p->ro, buf + sizeof(unsigned long) + name_len + 1 + sizeof(uint8_t), sizeof(uint8_t));
23   *param = p;
24   return sizeof(unsigned long) + name_len + 1 + 2*sizeof(uint8_t);
25 }
```

Figure 44: C implementation.

itself (**SerializedName**), the parameter's type (**Type**), its value (**Value**), and a boolean indicating whether the parameter is read only (**Ro**). The decoded information is used in line 14 to build an Erlang record, prepended to the

```
1  % Decode Payload by calling the two-args version of the function passing an empty list,
2  % which will be filled with the data extracted from the payload
3  decode_params(Payload) -> decode_params(Payload, []).
4
5  % Pattern matching on the first arg: if the binary variable is empty, then we finished
6  % (we reached the base case for the recursion) and we can return the ListOfPars...
7  decode_params(<<>>, ListOfPars) -> ListOfPars;
8  % ... otherwise, the first byte (L1) contains the length of the parameter's name (next field),
9  % and the following bytes represent: its type, its value, and it being read-only; the rest
10 % of the payload contains other parameters that will be extracted in the next (recursive) call
11 decode_params(<<L1:8, SerializedName:L1/binary, Type:8/unsigned-integer,
12     Value:8/unsigned-integer, Ro:8/unsigned-integer, Rest/binary>>, ListOfPars) ->
13     % Fill a new record with the extracted content
14     NewRecord = #parameter{name = erlang:binary_to_list(SerializedName),
15         type = Type, value = Value, ro = Ro},
16     % Recursive call to continue parsing the payload. The new record is saved into the list
17     decode_params(Rest, [NewRecord|ListOfPars]).
```

Figure 45: ELIoT implementation.

list of decoded parameters during the recursive call in line 17. Overall, the 25 lines of C code in Figure 44 reduce to 7 lines of (uncommented) ELIoT code in Figure 45.

```
1  [...]
2  char msg = 'M';
3  memset(&destAddr, 0, sizeof(struct sockaddr_in));
4  destAddr.sin_family = AF_INET;
5  destAddr.sin_port = htons(PORT);
6  destAddr.sin_addr.s_addr = destIp;
7  sock = socket(AF_INET, SOCK_DGRAM, 0);
8  setsockopt(sock, SOL_SOCKET, SO_BROADCAST, (void *) &broadcastPermission,
9             sizeof(broadcastPermission));
10 sendto(sock, &msg, 1, 0, (struct sockaddr *) &destAddr, sizeof(struct sockaddr_in));
11 [...]
```

Figure 46: C code for sending beacon messages in the smart-home application—functionally equivalent to lines 7 and 10 in Figure 43.

Similar benefits are found in creating and sending messages. For instance, Figure 46 shows the C code necessary to prepare and broadcast a beacon message, as done in lines 7 and 10 of Figure 43. The tedious code necessary to setup the UDP socket and the broadcast address are replaced in ELIoT by addressing to **`all`** and the ~ operator. This makes the 9 lines of C code shrink to only 2 in ELIoT.

Generally, one might argue that the more compact implementations attainable using the functional paradigm lead to higher chances of programming errors, essentially because the code is semantically more dense. The evidence, however, demonstrates that this is not the case. On the contrary, and especially for highly distributed functionality, the more compact code resulting from the use of functional programming ultimately yields more dependable systems [29, 30].

Because of its Erlang core, ELIoT also simplifies implementing concurrent functionality, by virtue of dedicated language and system support to multithreading. As an example, mutexes and condition variables, required in C to explicitly synchronize concurrent threads, are unnecessary with ELIoT. Already in the relatively simple smart-home application, nonetheless, C programmers heavily rely on such synchronization primitives to coordinate ac-

cess to the shared list of appliances. ELIoT programmers can, on the other hand, organize the code in such a way that the list of appliances is modified by the receiving thread only, whereas other threads operate on an immutable copy of such data structure, included in the message that triggers their processing.

6.2 QUANTITATIVE EVALUATION

Previously, we introduced the hardware devices we ported ELIoT to; this evaluation considers three of them:

- a Raspberry Pi board (ARM processor), equipped with 256 MB of RAM
- an ebook reader (Kindle 4th gen., ARM processor), equipped with 256 MB of RAM
- a custom embedded board with a RT3050 MIPS processor called "Carambola", featuring 32 MB of RAM.

In particular, the ebook reader has been tested mainly for its power saving features, while the other two devices have been thoroughly tested for CPU and memory consumption, too.

6.2.1 *System Performance*

Increasing developers' productivity often comes at a cost. This is also the case for ELIoT, where such cost materializes as performance overhead. To precisely evaluate this aspect, we compare the performance of the C and ELIoT implementations of the smart-home application by measuring memory consumption, CPU usage and power consumption, as well as network traffic. We perform this comparison on both embedded devices currently running the ELIoT VM.

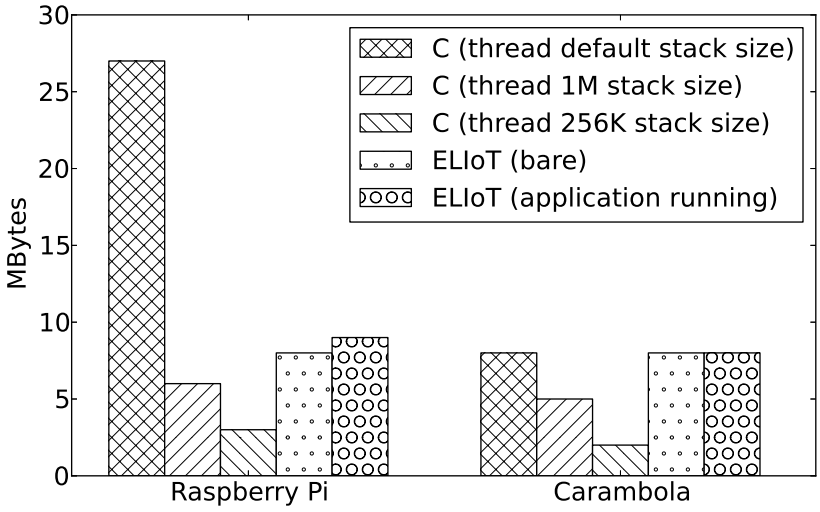


Figure 47: Memory consumption (pmap).

Memory

We measure memory usage with *pmap*: a Linux utility that reports the entire memory allocated for a given application, including code, libraries, stack, and heap. This gives a precise indication of the amount of memory a device needs to run the application: devices with less memory would just be unable to run the same application implementation.

Figure 47 reports the results. The caveat in the results we obtain from the C implementation is that it uses the *pthread* library for multiprocessing, which leaves to programmers the burden to explicitly choose the stack size for each thread. Over-provisioning this value is common practice in mainstream programming, as plenty of memory is typically available. In embedded system programming, however, this is conducive to interesting observations: a naive C programmer who uses the default stack size[17] would build an application

17 The default stack size in the *pthread* library is 8 MB for the Raspberry Pi (vanilla Linux) and 2 MB for Carambola (OpenWrt).

that uses the same or more memory than the corresponding ELIoT implementation. ELIoT programmers, on the other hand, rely on lightweight multiprocessing provided by the VM and do not need to worry about such system configuration details. Nevertheless, a skilled C programmer able to manually fine-tune the system configuration—a typically error-prone and time-consuming task—would find working settings at 1MB or even 256 KB per-thread stack space, the latter being the minimum that allows the application to run correctly. In this case, the C implementation consumes less than half the memory of the ELIoT implementation.

To better characterize memory usage in ELIoT, we separately assess the VM with no application loaded and when the smart-home application is running. As shown in Figure 47, it turns out that the VM is responsible for most of the memory used by ELIoT, with the application requiring only a few additional KB. This has two consequences: *i)* it clearly points at the VM as an avenue for further improvements to battle the memory overhead in ELIoT; and *ii)* it suggests that the gap between C and ELIoT likely reduces with more complex applications, as the memory occupation due to the VM is a fixed cost paid once and for all.

CPU usage and power consumption

We measure the time the CPU is busy processing using the *getrusage* primitive, which returns per-process CPU time split between user and system time. At the control panel, we run 50 consecutive executions of the operations to compute the appliances' schedule, as per functionality **F2**, by assuming that the expected energy consumption at the appliances is computed remotely, corresponding to scenario **A**. We also include six rounds of beaconing for discovery and monitoring of appliances between scheduling operations, as per functionality **F1**. Such setting is representative of foreseeable usages of the smart-home application. Each cycle lasts 60 seconds. We repeat the 50 iterations across 30 different runs, and plot the resulting average with the 95% confidence intervals.

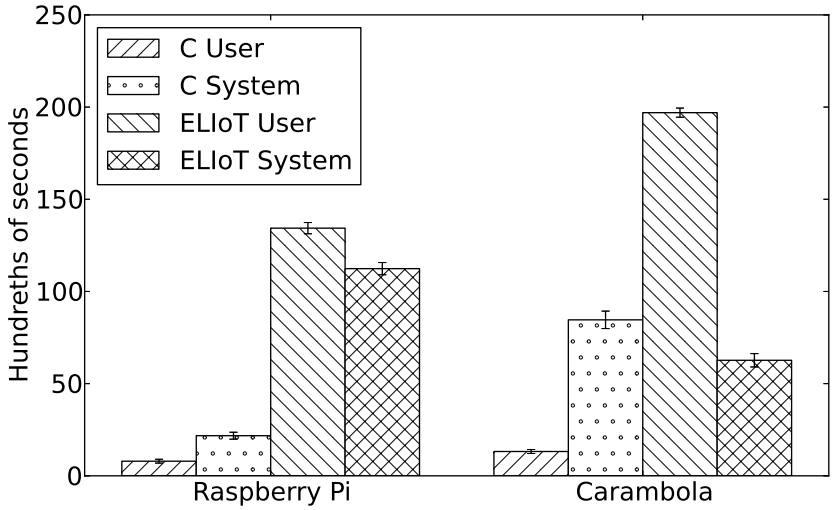


Figure 48: CPU times.

Figure 48 depicts the results. Using the C implementation, the user time is much lower than the system time, especially on a relatively powerful device like the Raspberry Pi. Differently, the time spent by the CPU using ELIoT on the Raspberry Pi is split almost equally between user and system time, while on the Carambola most time is spent executing user code. Using ELIoT, both user and system times are larger compared to the C counterparts. In absolute terms, however, the latencies that such CPU times may introduce are less than 30 ms per iteration, which includes a schedule computation and six rounds of beaconing. These are reasonably within tolerance of non-realtime applications such as a smart-home.

Increased CPU times also correspond to higher power consumption. To assess this aspect, we hook the Raspberry Pi and the Carambola to a professional voltage generator/multimeter to measure their average power consumption throughout a single application iteration.

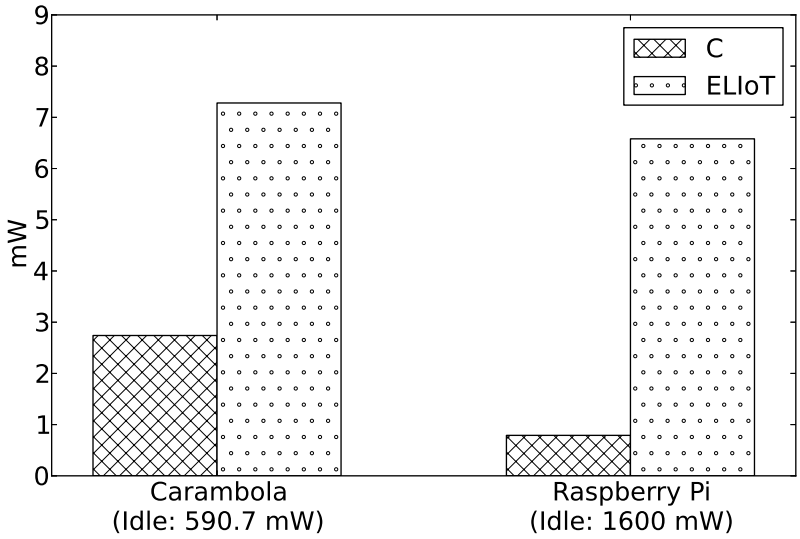


Figure 49: Power consumption. (The idle power consumption is factored out.)

Figure 49 shows the results of our measurements by factoring out the power consumption when the board is completely idle. Compared to the C implementation of the smart-home application core functionality, ELIoT imposes an overhead of about 5 mW on the Carambola and of 6 mW on the Raspberry Pi, arguably negligible for the scenarios we consider. Adding the idle baseline to the measures above results in a relatively high overall figure for the platforms we tested, which are not optimized for limiting power usage. On the other hand, better engineered platforms exist, which are powerful enough to run ELIoT and still have a reduced power usage, in particular at idle. For example, a modern smartphone using a Samsung S3C2442 SoC absorbs about 268 mW when idle [31], while the ARM board that runs the Amazon Kindle 4 (a device explicitly designed for low power consumption) absorbs 45 mW when idle with wifi enabled and connected, as we measured using the same equipment used for the other platforms.

Figure 50 shows the previous chart with this additional device: in this case, the idle power consumption is (more than) one order of magnitude less than the other two, at the same time the difference between idle and full throttle is more noticeable, due to the fact that the Linux version running on the Kindle is compiled with *soft floating point*, which means that part of the floating point calculations have to be emulated in software (on the other hand, boards such as the Raspberry Pi use *hard floating point*, taking advantage of the full power of hardware arithmetical units).

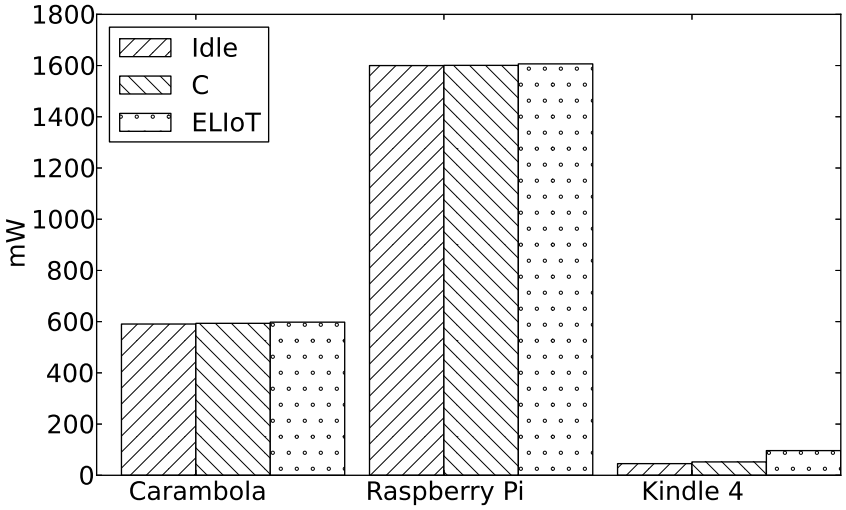


Figure 50: Total power consumption, with Kindle.

Network traffic

Using a standard network inspection tool, we measured the amount of bytes transferred through the network during a single application iteration. This includes several messages exchanged between the control panel and the appliances. The application payload is the same for both the C and the ELIoT implementation.

Table 8: Network traffic.

System	Bytes
C	1929
ELIoT	2126
Overhead ELIoT (%)	10.21

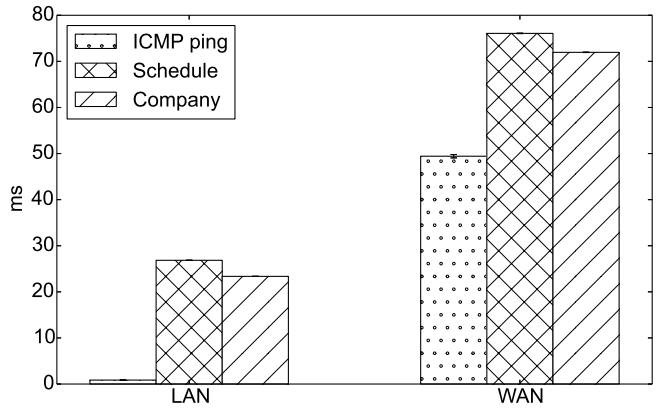


Figure 51: Network delay (LAN and Internet).

Table 8 shows the results.

The total overhead of ELIoT with respect to C is 10.21% (2126 vs. 1929 bytes). This is due to the small additional header that the ELIoT VM adds to every message to support the abstract addressing mechanisms, e.g., to reach specific ELIoT processes within a given node. The *number* of messages, however, is the same in both the implementations. The small overhead due to ELIoT is then still practical.

We also measured the network latency in Internet-scale interactions, to show how the framework performs when ELIoT messages move in a Wide Area Network, a typical case for IoT applications.

Figure 51 shows the round-trip time of two exemplary network messages used in our scenario (the "schedule" message is a single step of the interactions required to prepare the appliances' schedule, while the "company"

message is one interaction from the electrical company to the control panel. They are both representative examples of complex messages that can happen in an ELIoT application.

The LAN measurements have been taken using two Raspberry Pis on a wired connection to the same router, while the WAN measurements have been taken using two Raspberry Pis situated in two different countries (Italy and Sweden, in particular). The two devices were executing a control panel and an appliance for the first type of messages, and a control panel and a company for the second type. The chart shows also the network delay from ethernet to ethernet ports using a sequence of ICMP ping messages. In all cases, the interactions have been executed 10000 times, and the plot shows also the 95% confidence intervals, barely visible since they are under 0.5%.

It can be observed that, regardless of the network delay, it takes about 20 to 25 ms for each ELIoT message to traverse the network stack (from the application level down to the ethernet port) on the sender, traverse back the stack on the receiver, elaborate the response and send it back to the sender.

Spawn time

We assess the time needed by ELIoT to spawn a new process whose bytecode comes from the network. This is key to evaluate the actual usability of the ELIoT mechanisms to upload new functionality on a running node; for example, in the smart-home application when appliance manufacturers need to update the on-board software. In particular, we measure the time it takes from when a message with the necessary bytecode is received at the node to when the new functionality is ready to accept input data. On average, this time goes from 50 ms on the Raspberry Pi to less than 20 ms on the Carambola (Figure 52): arguably acceptable in most practical scenarios.

6.2.2 *RESTful interfaces performances*

In Section 2.2, we presented the CREST-Erlang framework and the performance evaluation of that implementation, compared with the original version

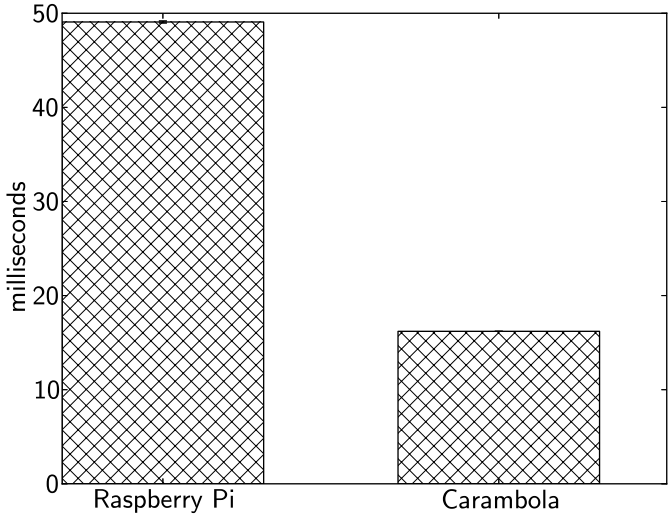


Figure 52: Spawn time

of the framework. In particular, the Erlang implementation was three orders of magnitude faster than the original Scheme implementation, and the overhead introduced by the CREST-ERLANG framework only slightly deteriorated the performances with respect to a standard REST application built with the same language (less than 10 ms in response time and a small increment in throughput).

As described earlier (Section 4.4), we used the experience from the development of the CREST-ERLANG framework to implement dynamic RESTful interfaces for ELIOT, confident in the small price in terms of performances that Erlang Web frameworks introduce, due to the parallelization characteristics of the language itself. The next two charts will show how the integration of these interfaces in ELIOT impact its performance.

The first chart, in Figure 53, shows the spawn time of a new REST function, using either the Web manager or ELIOT API, which takes about 130 ms, and the REST invocation of the same function , which takes about 20 ms; the REST

overhead doubles the spawn time of native ELIoT functions, as shown on Figure 54: this is the time taken by the framework to unpack the HTTP request, spawn the process and return the HTTP response to the caller. Both the server and the client were connected on a wired LAN, and the measurements were taken on the Raspberry Pi device.

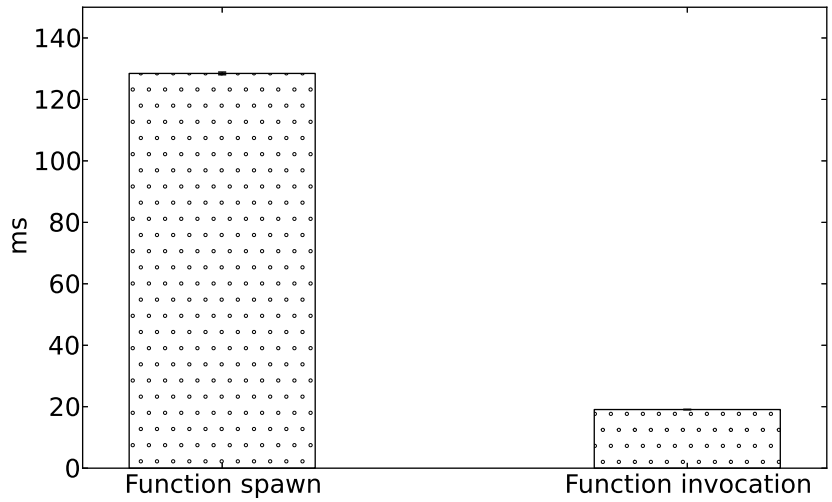


Figure 53: Function spawn absolute timings on Raspberry Pi

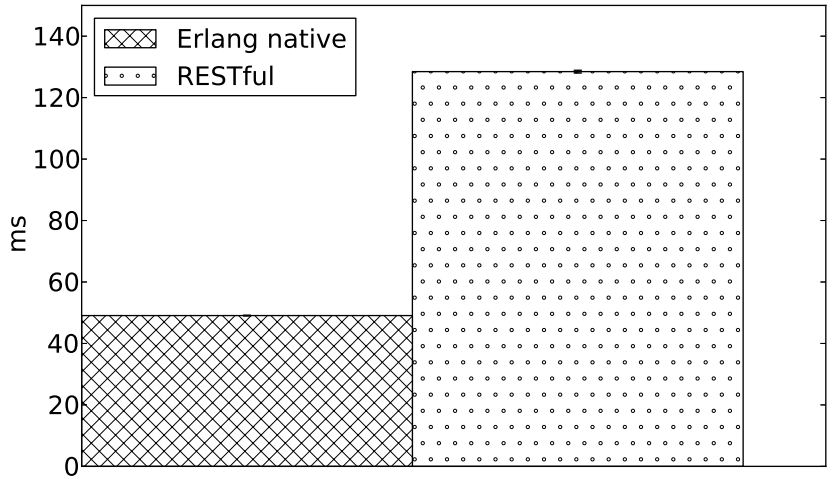


Figure 54: REST spawn time with respect to ELIoT spawn time

7

RELATED WORKS

This chapter will illustrate existing works that include both sides of IoT programming, enabling local or remote communication and spanning the entire spectrum of devices described in the previous chapter. This will help point out the differences with respect to ELIoT.

We will describe the problems in developing applications in IoT scenarios, and the difficulties in testing and debugging that have been underlined by several research works.

We will then show existing works similar to the Computational REST architecture and paradigm, a contribution that has been described on Chapter 2 and that has been the foundation for the dynamic RESTful interfaces offered by ELIoT and described in Section 4.4.

Finally, we will illustrate existing works to apply formal verification to embedded distributed applications, and how they differ from the model checking tool described in Section 5.2.

7.1 SENSOR NETWORK PROGRAMMING AND PERVASIVE COMPUTING

The first side of IoT programming is also the "older" one: the Wireless Sensor Networks research field is at least a decade old, and the local interactions paradigm is an intrinsic part of its programming model.

Existing solutions for sensor network programming [16] allow an efficient implementation of localized interactions by deploying the application logic right onto the embedded devices; at the same time, these solutions have several limitations, as described in [16] and [32].

Level of abstraction

The first limitation is that these frameworks and operating systems offer a very low level of abstraction, where the developer has to know all the intimate details of the hardware platforms that she is targeting, to be able to fine-tune the application settings and set the algorithms that she will use, especially regarding the network communication. This generates ad-hoc solutions for specific problems, and makes it difficult to generalize and reuse at least parts of the developed software.

This is well represented by two well-known operating systems for Wireless Sensor Networks, TinyOS [2] and Contiki [3]: the former is an event-based open source operating system, able to run on several typical low-level devices used in WSN, while the latter is another open source operating system with a programming model more similar to standard programming languages. Both these systems are programmed through (a dialect of) C, and require knowledge of the platform and offer a very detailed API to control each hardware component. This means that changing the hardware usually means changing parts of the application, possibly re-thinking the algorithms to accomodate the new hardware characteristics.

Things are even more difficult with commercial or proprietary operating systems for WSNs, such as WaspMote [33], which offer their own specific APIs and are in general difficult to port to different platforms, or integrate into competitive products.

Other systems try to avoid these problems by offering a higher level of abstractions, to help portability and reuse of the software. For example, Regiment [34] features a functional programming model, providing primitives such as *fold* and *map* to process data originating from subsets of nodes. Flask [35] provides a data-flow programming model based on discrete computational steps, akin to side effect-free function calls. Snlog [36] 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 [16]—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 tries to offer advantages of both these approaches: on one hand, it uses a high level programming language, with typical functional programming mechanisms useful for manipulating streams of data coming from sensors; being VM-based, it hides to the developer the low-level layers of the application, but it allows the developer to easily access parts of these layers, for example letting her choose whether to use or re-implement reliability mechanisms; the developer can also access parts of the VM implementation and "plug in" her own implementations of network distribution protocols and hardware interfaces, depending on the deployment devices.

Local communication

The second limitation is that it is largely common to these approaches the view of the sensor network as a stand-alone system, where Internet-scale interactions are at best mediated by ad-hoc gateways that are to be designed and implemented on a per-application basis. Different solutions provide remote access to sensors and actuators from the Internet, such as sMAP [37] and CoAP [38], that implement low-powered IPv6 networking, but the limitations of these kinds of devices limit the usefulness of these protocols: adding a full network stack to these devices usually means moving to the "remote communication" paradigm and render difficult the additional implementation of a local communication.

From a conceptual standpoint, ELIoT aims at bringing the localized interactions, already enabled by sensor network programming, in Internet-connected embedded networks.

Testing and debugging

Finally, worth noticing is that sensor network operating systems often come with an accompanying simulator, e.g., TOSSIM [25] for TinyOS and Cooja/MSPSim [39] 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 [40], to avoid emulation of the real devices and preferring the execution of the application on real instances of them, at the same time in a simulated environment which allows testing of scalability other than hardware interfacing.

7.2 IOT ARCHITECTURES AND APPLICATION FRAMEWORKS

Significant activities are undergoing to define software architectures for the IoT, spanning from the network to the application layer. For example, the IoT6 project [41] exploits an IPv6-based network layer to build CoAP services atop. The IoT-A project [42] defines an architectural reference model for the interoperability of IoT devices, whereas Spitfire [43] investigates unified concepts for facilitating the effective development of IoT applications.

ELIoT is largely complementary to these efforts. Sound software architectures are necessary to improve interoperability, organize applications' functionality, and reason about the system operation. Orthogonal to these aspects is how to specify the actual application processing within the individual components, and how to establish their distribution across the networks of sensors and actuators, and Internet-side services. ELIoT provides effective support for the latter aspects.

In terms of distributed coordination, integrating smart objects with the Internet may follow two communication models. Solutions exist to proactively export sensor data to the Internet, such as Publish/Subscribe middleware [44], shared memory systems [45], and platforms providing storage and processing facilities for sensor data, such as Cosm [46].

At a logical level, in both approaches the application logic runs outside the network of embedded sensor and actuators. This simplifies prototyping IoT applications, yet it does not allow an efficient implementation of combined Internet-scale and localized interactions. ELIoT aims at efficiently enabling the latter by retaining the ability to coordinate with Internet-side services. For example, as seen in the smart-home scenario, ELIoT developers can im-

plement control loops that span neighboring devices and integrate them with externally-running services.

Similar considerations apply in relating ELIoT to the body of work in pervasive computing systems. For example, Aura [47] and Gaia [48] focus on effective development of interactions between users and the devices they operate: the former offers a system trying to understand the behavior of its users, and leverage this knowledge to modify the office or home behavior to accomodate the current user task; Gaia offers a similar system, where *virtual spaces* created by users are associated to real spaces and real devices, again to accomodate user tasks in the environment. Since both these platforms offer high level functionality, the underlying operating system has to integrate as many different devices as possible, and a framework like ELIoT could easily offer this kind of portability by leveragin its being based on a Virtual Machine.

MundoCore [49] provides a low-level framework and middleware for developing platforms integrating different devices, from mobile systems to computers in a homogeneous framework. Although MundoCore caters for effective integration of heterogeneous hardware, an issue we also tackle in ELIoT using a VM-based execution model, these system do not focus on how to effectively develop Internet-scale and localized interactions within the same application.

There also exist works tackling the development process of IoT applications. Srijan [50], for example, presents a model-driven approach by establishing specific roles for the involved stakeholders, and by introducing domain-specific languages (DSLs) to model both the application and the underlying systems. Interfaces and component connectors are automatically generated based on such models. Similar works are largely complementary to ELIoT, which focuses on providing effective programming and system support. For example, ELIoT may serve as a target language for Srijan, likely simplifying code generation.

7.3 APPLICATION SCENARIOS

Several researches have proposed many different "Internet of Things" scenarios to exemplify the type of interactions that can be obtained by this paradigm; an exaustive list of these proposals is for example [15], which has been sponsored by a company working on IoT and M2M interactions: we decided to exemplify the use of ELIoT on a smart-home application, inspired by both the *smart grid* and remote control appliance scenarios shown in the survey (n. 15 and n. 47).

Ad-hoc solutions exist for developing software in specific application domains. For example, Gator Tech [51] presents the design of a pervasive computing system especially conceived for elderly people, within an environment enriched by sensors and actuators; whereas HomeOS [52] is a middleware layer implementing higher-level abstractions for smart-home applications, giving the illusion that the house itself can be treated as a single computing device.

ELIoT's applicability extends beyond this particular context. For example, in the logistics domain, sensor attached to packages may provide fine-grained continuous monitoring of the shipped goods, used to take smart routing decisions and to inform business analysts at the back-end of item availability and market trends [53]. Such applications feature similar combinations of localized and Internet-scale interactions as our smart-home example. ELIoT precisely aims at enabling both kinds of interactions within the same development framework.

7.4 COMPUTATIONAL REST

Computational REST, the framework used as a foundation for the dynamic RESTful interfaces implemented in ELIoT, is related with current research on evolvable and dynamically adaptable software architectures and on programming languages supporting dynamic adaptation. Seminal work on the identification of the critical architectural issues concerning run-time evolution is

described in [54–56]. The CREST approach is largely motivated by this work. Several alternative architectural styles exist to support dynamically evolvable distributed applications. Hereafter we briefly review the most relevant ones and we contrast them with CREST.

Publish-subscribe (P/S) [57, 58] is an event-based style where components are not directly connected, but communicate through a common middleware system, which takes any new event notification and dispatches it to any component subscribed for that specific event. This structure is highly dynamic since nodes may be added and removed while the system is running; communication is asynchronous and components can operate independently of each other.

Map-reduce (M/R) [59] is a style used to parallelize a computation over a large data set by distributing work over a collection of worker nodes. In the *map* phase each node receives from a master node some amount of data and elaborates it, returning key-value pairs to the master, while in the *reduce* phase the master node takes the answers to all the sub-problems and combines them to produce the output. Because worker nodes may be masters, a tree structure can be easily obtained, increasing scalability. As in the P/S case, M/R nodes are completely autonomous; they may join and leave dynamically as they do not share any data or state directly, and perform their computation in isolation w.r.t. the others.

Similarly to CREST, P/S and M/R architectural styles are oriented to dynamic adaptation, but differently from CREST they are not specifically oriented to supporting Web applications, probably the most important domain for distributed applications today and the one we target.

The two architectural styles that are today competing for becoming a standard in building Web applications are REST and the *Service-Oriented architecture* (SOA) [60]. We already discussed the differences between CREST and REST in Section 2.2. SOA models a Web application as a composition of different autonomous services, independently developed and existing in different namespaces and execution contexts. Services may be dynamically discovered and compositions may bind to them dynamically. Usually these services op-

erate over HTTP using Web Service protocols supporting standardized discovery and service invocation. Unfortunately, these protocols violate REST principles, and this can be a major problem, since REST principles are those that guaranteed the success of the Web.

CREST not only follows the REST principles, but also promises to support dynamic adaptation much better. Indeed, both REST and SOA focus on data as the primary element exchanged among components and this makes it hard to adapt the architecture of the application dynamically, since this usually requires to introduce new components/services. Vice-versa, CREST adopts the computations themselves as the elements exchanged among nodes (i.e., peers) and this makes it straightforward to change the architecture of the application at run time, when required.

Besides architectural styles, another research direction related with the Computational REST work concerns programming languages. In particular, the identification of features or language constructs that may provide better support to the specific requirement of run-time adaptation. This sometimes leads to extensions of existing languages to support dynamic adaptation. For example, *context-oriented programming* extensions have been proposed and implemented for various languages [61], starting from initial work on LISP [62], up to the initial version of ContextErlang [63] developed by our research group. The features supported by Aspect-Oriented programming languages [64], and in particular Dynamic Aspect-Oriented languages [65], have also been proved to help in this context.

Functional programming languages, and in particular the notions of continuation and closure, have also been revamped in the context of Web programming. A short summary of work upon which CREST-Scheme is rooted can be found in [66], while examples of use of functional programming concepts in Web applications are provided in [67, 68].

7.5 MODEL CHECKING

Several works have been proposed during the years to apply model checking techniques to embedded distributed systems, in particular to Wireless Sensor Networks, with different purposes: to model check this kind of applications, or specific algorithms, or properties of the typical network layers in use (e.g.,, to compare variants of the Medium Access Control of IEEE wireless protocols).

Anquiro [69] refers to the first kind: it is a tool for model checking WSN applications. It is based on the Bogor [70] model checker and it provides different levels of abstraction to overcome the problem of state space explosion:

- hardware specific, to verify low-level functionality
- neighborhood communication, to verify network level functionality independently of the hardware platform
- system-wide communication, to verify application level functionality independently of the network layer.

The tool translates C code implemented using the Contiki OS [3], and then verifies the model obtained by the translation. Users can specify properties using Linear Temporal Logic, and the tool searches for counter-examples to these properties, which can then be inspected by the developer.

ELIoT model checker relies on similar concepts: it verifies applications written using the ELIoT framework, without specific translations to different modeling languages, due to the fact that it uses a model checker written in the same language as ELIoT applications; properties can be specified using LTL, and the tool is used to verify application level functionality, with network models that can be tuned by the developer, depending on the type of environment in which the application will be deployed.

T-Check [71] is a tool for the TinyOS framework [2] to detect violations of properties about the system. The tool starts from the application source code, enhanced with assertions and properties used to reason about the system. Users can specify safety properties (all states of the system must not violate

them) or liveness properties (they must be true at least in one of the states). The tool is built upon the TinyOS simulator, TOSSIM [25], which is driven by scripts written with programming languages different from TinyOS C dialect (i.e., Python), making the tool able to work with high-level properties, losing the low-level specific issues.

ELIoT model checker does not need assertions, but it needs at least one "probe" specified in the system, so that the tool knows then a certain property holds; as mentioned before, it builds upon the same programming language of the applications, thus allowing developers to verify properties spanning their entire application.

The problem of using model checkers written in different languages with respect to the applications can be observed also in [72], where the authors modeled TinyOS applications using hybrid automata, able to capture both discrete transitions and continuous flow transitions; these models fit quite well for TinyOS applications, where parts of the system (e.g., energy consumption) can be modeled using continuous variables, and the operational behavior is modeled on finite state automata. The model languages used here are HyTech [73] and SHIFT [74], thus the tool relies on the precision of the translation of the application.

The same considerations apply also to [75], which is a work that models the *Optimal Geographically Control Algorithm* by using Real-Time Maude [76], which is a model checking tool that can be applied to real-time systems: in this case, the algorithm has been specified by using this modeling language and analysed using this tool, thus relying on the precision of the translation from the original algorithm to the model, obtaining results that could be quite different from the ones obtained analysing a specific implementation of the same algorithm. Again, ELIoT overcomes these problems by using the same programming language.

A characteristic of the ELIoT model checker is the fact that is uses deterministic model checking techniques, providing probabilistic informations to the developer to inspect the behavior of her application. There have been works that used probabilistic model checking, for example to check prop-

erties of the Medium Access Control of IEEE 802.11 protocols using probabilistic timed automata and UPPAAL [77]; similar works verified properties using Discrete Time Markov Chains (again, probabilistic models) and PRISM for IEEE 802.15.4 protocols used in WSNs [78]. In both cases, the researchers wanted to demonstrate specific properties of the wireless protocols, from maximum intervals between successive correct packet transmissions to energy consumption for low power protocols. In both cases, their aim was at a lower level with respect to ELIoT purpose.

8

CONCLUSIONS

This thesis has presented ELIoT, a new framework for developing applications for the "Internet of Things".

The work started from a broad analysis of the existing frameworks used to develop IoT applications with the corresponding devices, from Wireless Sensor Networks' low-powered boards, equipped with simple microcontrollers, to today's smartphones and tablets, equipped with multicore processors and powerful communication capabilities.

We have described the two main paradigms under which we can classify these frameworks, namely *local communication enablers* and *remote communication enablers*. The former provides APIs and capabilities allowing local communication between devices, a paradigm common in Wireless Sensor Networks, where the application-level decisions are taken *inside* the network and gateways to the Internet are used mainly to send data to online collectors for further (offline) analysis. The latter provides APIs to easily communicate with remote devices and computers through the Internet, for example with cloud systems, to collect data and take decisions. In both cases, it is not easy to perform the complementary action: for example, Wireless Sensor Networks operative systems seldom offer the capability to integrate into the Internet and directly communicate with Web services. At the same time, frameworks for more powerful devices (e.g., smartphones) do not offer APIs to enable local communication: if two of these devices have to exchange data, the easiest way is to use services provided through the Internet, even if the two devices are next to each other.

We think that this can be changed, and we introduced ELIoT to show that it is possible to offer a powerful framework that allows both types of communication. In doing so, we leveraged some of the existing features of the

Erlang language and Virtual Machine, and introduced features for both types of communication:

- different network distribution protocols, which give more control to the developer on how the VM behaves when sending messages to remote processes. In particular, ELIoT offers specific mechanisms to make the transmission reliable or non-reliable, and the developer can decide to improve upon these options depending on the type of the application. Moreover, the framework provides broadcast communication with the possibility of disseminating computation, using context filters to select only parts of the network. We also simplified the network distribution protocol used by the VM, in particular removing the overlay network between Erlang nodes[18], giving the application developer the responsibility of creating and maintaining an overlay only when needed
- the capability of deploying computations at runtime and automatically generate REST interfaces to invoke them, so that they can be easily integrated into different frameworks and used outside ELIoT (even browsed through the Internet, if necessary). This capability leveraged our knowledge of Web services in Erlang, due to the development of a previous framework to change the Web of resources to a Web of computations (the Computational REST paradigm and architecture).

At the same time, testing and debugging IoT applications is known to be quite difficult, for example to correctly integrate a model of the environment or to test algorithms scalability. This is due to the peculiarities of the applications and the lack of sound software engineering tools and procedures: for low-powered devices, the developer relies on her knowledge of the low-level aspects of the boards and the operative systems, creating a solution specific to the targeted scenario. On the other hand, applications for more powerful devices (e.g., smartphones) are developed with frameworks that offer simulators unable to reliably emulate the hardware platform, especially for those

18 Instances of the VM running on a device.

operating systems (e.g., Android) offering a wide variety of different devices (smartphones, tablets, TV accessories).

ELIoT provides a simulator tool that tests the code *as is*, emulating the network and the environment using known tools and allowing the developer to fine-tune the environment model (to the extent of plugging in new model implementations), leveraging her knowledge of the deployment situation. Moreover, the simulator allows mixed scenarios, where part of the network is simulated and part runs on real devices: this avoids having to emulate the hardware of several different devices, and obtain more precise measurements from the real hardware.

ELIoT also provides a modified version of an existing model checker for Erlang (called McErlang [23]), to reflect the peculiarities of IoT scenarios and the mechanisms that ELIoT provides with respect to "vanilla" Erlang. The developer can verify LTL properties of the application, obtaining validation proofs or counter-examples violating those properties. This tool is written in Erlang, and this means that it is able to verify ELIoT applications without changing the code or having to translate it into a different modeling language, thus producing more reliable results with respect to solutions that need a model translation.

Finally, we tested the ELIoT framework on several different devices, going from small, WSN-like boards, to smartphones and more powerful devices. We tested a ELIoT application against a functionally-comparable C version, using a typical IoT scenario as a starting point, and compared the performances of these two versions on some of these devices: the ELIoT version performed as the C one from a memory consumption point of view, while having worse performances from a CPU point of view. This is due to the fact that ELIoT applications run on a Virtual Machine, and a VM it requires more CPU time, but at the same time allows developing more portable programs (the same application was tested on MIPS, ARM and Intel microprocessors without changes in the code, even when interfacing with external hardware). We also tested ELIoT on special devices such as ebook readers, to show how low can the energy consumption be also for ELIoT applications (despite the existence of

a VM and the need of more CPU time), if the device has been developed with energy saving in mind[19].

8.1 FUTURE WORK

Future work on ELIoT will involve different parts of the framework: from Virtual Machine improvements, to empowered RESTful interfaces, to thorough testing of the hardware side and the verification tool.

The Virtual Machine and the standard library

The main limitation of ELIoT is the design of the existing Erlang Virtual Machine: we simplified the functionality of the existing Erlang network distribution protocol and removed not used libraries, but its footprint, especially from a memory point of view, is still quite high. We want to investigate how to improve this situation, with possible collaborations with other Erlang developer groups that are working on running the language and platform on embedded devices.

We want to investigate how to improve the filtering capabilities of broadcast **`spawn`** operations: as mentioned in the previous chapters, an important limitation is the fact that transmission of the bytecode is needed to send complex filtering functions to remote nodes; we would like to integrate such solutions in a seamless way in the Virtual Machine, to expand the filtering options given to the developers.

RESTful interfaces

ELIoT introduced dynamic RESTful interfaces and the ability of automatically wrap modules and applications into Web services, that can be invoked through standard HTTP calls.

New Web standards, such as HTML5, have introduced new interactions between clients and servers using HTTP operations, such as Web sockets and

19 Which is not always the case, for example for devices such as the Raspberry Pi.

asynchronous communication; moreover, AJAX technologies and libraries help obtaining a better user experience when using Web applications. We want to introduce some of these technologies into our dynamic RESTful interfaces, to improve the Web manager, introduce JSON calls to Web services and limit the need to handle HTTP timeouts by the developers, making this issue transparent to module invocations.

Hardware testing

The IoT world is in continuous evolution, and new devices with new peculiarities or improvements with respect to previous ones are presented almost every day; our effort is to test ELIoT on all the devices that seem interesting to us and that we think can provide benefits to the developers.

Moreover, we already support ZigBee network protocols, and we want to introduce a library to seamlessly communicate with Wireless Sensor Networks and interface with devices that, due to hardware limitations, cannot run ELIoT. We already developed a specialized library for low-powered devices to show how to handle the network protocol on non-Erlang devices (see Section 4.2.2) and remain compatible with at least part of the ELIoT network messages; we want to integrate similar libraries on WSN operating systems like TinyOS.

Model checking

We introduced McErlang as a model checker for Erlang applications, and discussed the modifications we made to integrate it with ELIoT functionality and with developer needs in IoT scenarios; we are now in the process of comparing real world data with results obtained in our tests, to improve the instruments the tool gives to the developers for fine-tuning verification properties.

We are also testing the complexity of model checking applied to large applications, to show the limitations in terms of LTL properties and state space explosion that the model checker is able to stand, and how to reduce the num-

ber of states in parts of the execution tree that can be ignored during the state space exploration.

8.2 FINAL REMARKS

This thesis has shown the benefits of supporting multiple communication paradigms for "Internet of Things" applications, and the correspondence between distributed mobile/embedded applications and the actor model supported by Erlang. It also showed the benefits of having tools to support the development of applications in these complex scenarios.

We think that different programming models, paradigms and tools are needed to thoroughly explore this exploding area of computer science and engineering and to produce robust and reliable applications, especially because these applications are integrating into everyday devices and our lives more and more every day.

BIBLIOGRAPHY

[1] Openenergymonitor. http://openenergymonitor.org.

[2] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, November 2000.

[3] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. Int. Workshop on Embedded Networked Sensors*, 2004.

[4] Alessandro Sivieri, Luca Mottola, and Gianpaolo Cugola. Drop the phone and talk to the physical world: Programming the internet of things with Erlang. In *SESENA*, pages 8–14, 2012.

[5] Alessandro Sivieri, Luca Mottola, and Gianpaolo Cugola. Eliot: Building internet of things software combining localized and internet-scale interactions. *Major revision submitted for publication on Computer Networks*, 2013.

[6] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. URL http://portal.acm.org/citation.cfm?id=1624775.1624804.

[7] Justin Ryan Erenkrantz. *Computational REST: a new model for decentralized, internet-scale applications*. PhD thesis, Long Beach, CA, USA, 2009. Adviser-Taylor, Richard N.

[8] Alessandro Sivieri, Gianpaolo Cugola, and Carlo Ghezzi. Computational rest meets erlang. In *Proc. 49th Int. Conf. on Objects, Models, Components, Patterns*. Springer-Verlag, 2011. ISBN 978-3-642-21951-1.

[9] Roy T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. URL `http://portal.acm.org/citation.cfm?id=932295`.

[10] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24:342–361, 1998.

[11] Justin R. Erenkrantz, Michael Gorlick, Girish Suryanarayana, and Richard N. Taylor. From representations to computations: the evolution of web architectures. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the european software engineering conference and the 14th ACM SIGSOFT symposium on Foundations of software engineering*, pages 255–264, New York, NY, USA, 2007. ACM Press. ISBN 9781595938114. doi: 10.1145/1287624.1287660. URL `http://dx.doi.org/10.1145/1287624.1287660`.

[12] M.M. Gorlick, J.R. Erenkrantz, and R.N. Taylor. The infrastructure of a computational web. Technical report, University of California, Irvine, May 2010.

[13] Mochiweb. `http://github.com/mochi/mochiweb`.

[14] H. Farhangi. The path of the smart grid. *Power and Energy Magazine, IEEE*, 8(1):18 –28, january-february 2010. ISSN 1540-7977. doi: 10.1109/MPE.2009.934876.

[15] Libelium top 50 iot applications. `http://www.libelium.com/top\_50\_iot\_sensor\_applications\_ranking`.

[16] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Compututing Surveys*, 2011. ISSN 0360-0300.

[17] Koen Langendoen and Niels Reijers. Distributed localization in wireless sensor networks: a quantitative comparison. *Comput. Netw.*, 43(4), 2003.

[18] Scripting layer for android. URL `https://code.google.com/p/android-scripting/`.

[19] Netduino. URL http://netduino.com.

[20] Alessandro Sivieri. Velux-lab: Monitoring a nearly zero energy building. In *Proceedings of the Fifth Workshop on Real-World Wireless Sensor Networks*, 2013.

[21] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection Tree Protocol. In *Proc. Int. Conf. on Embedded Networked Sensor Systems*, 2009. ISBN 978-1-60558-519-2.

[22] Z. Shelby, K. Hartke, and C. Bormann. Constrained Application Protocol (CoAP), draft 18. RFC draft 18. URL https://datatracker.ietf.org/doc/draft-ietf-core-coap/.

[23] Lars-Ake Fredlund and Hans Svensson. Mcerlang: a model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 125–136, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291171. URL http://doi.acm.org/10.1145/1291151.1291171.

[24] Alex Bernauer and Kay Roemer. Meta-debugging pervasive computers. In *Proc. Workshop on Programming Methods for Mobile and Pervasive Systems*, 2010.

[25] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proc. 1st ACM Conf. on Embedded Networked Sensor Systems*, 2003. ISBN 1-58113-707-9.

[26] Kannan Srinivasan, Maria A. Kazandjieva, Saatvik Agarwal, and Philip Levis. The β-factor: measuring wireless link burstiness. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 29–42, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-990-6. doi: 10.1145/1460412.1460416. URL http://doi.acm.org/10.1145/1460412.1460416.

[27] Kannan Srinivasan, Mayank Jain, Jung Il Choi, Tahir Azim, Edward S. Kim, Philip Levis, and Bhaskar Krishnamachari. The κ factor: inferring protocol performance using inter-link reception correlation. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, MobiCom '10, pages 317–328, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0181-7. doi: 10.1145/1859995.1860032. URL http://doi.acm.org/10.1145/1859995.1860032.

[28] Kannan Srinivasan, Prabal Dutta, Arsalan Tavakoli, and Philip Levis. An empirical study of low-power wireless. *ACM Trans. Sen. Netw.*, 6(2):16:1–16:49, March 2010. ISSN 1550-4859. doi: 10.1145/1689239.1689246. URL http://doi.acm.org/10.1145/1689239.1689246.

[29] Ulf Wiger, Gösta Ask, and Kent Boortz. World-class product certification using erlang. *SIGPLAN Not.*, 37(12), 2002.

[30] Bruce J. MacLennan. *Functional programming: practice and theory*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[31] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proc. of the USENIX annual technical conference*, 2010.

[32] Gian Pietro Picco. Software engineering and wireless sensor networks: happy marriage or consensual divorce? In *Proc. FSE/SDP Workshop on Future of Software Engineering Research*, 2010. ISBN 978-1-4503-0427-6.

[33] Waspmote. www.libelium.com/waspmote.

[34] Ryan Newton, Greg Morrisett, and Matt Welsh. The Regiment macroprogramming system. In *Proc. Int. Conf. on Information Processing in Sensor Networks*, 2007. ISBN 978-1-59593-638-7.

[35] Geoffrey Mainland, Greg Morrisett, Matt Welsh, and Ryan Newton. Sensor network programming with Flask. In *Proc. Int. Conf. on Embedded Networked Sensor Systems*, 2007.

[36] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *Proc. Int. Conf. on Embedded Networked Sensor Systems*, 2007. ISBN 978-1-59593-763-6.

[37] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. sMAP: a simple measurement and actuation profile for physical information. In *Proc. 8th ACM Conf. on Embedded Networked Sensor Systems*, 2010.

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

[39] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. COOJA/MSPSim: interoperability testing for wireless sensor networks. In *Proc. Int. Conf. on Simulation Tools and Techniques*, 2009. ISBN 978-963-9799-45-5.

[40] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin. EmStar: a software environment for developing and deploying wireless sensor networks. In *Proc. USENIX Annual Technical Conference*, 2004.

[41] IoT6 - Universal Integration of the IoT. `www.iot6.eu`, .

[42] Internet of Things - Architecture. `www.iot-a.eu`, .

[43] Spitfire Semantic Web interaction with Real Objects. `spitfire-project.eu`.

[44] G.C. Fox, S. Kamburugamuve, and R.D. Hartman. Architecture and Measured Characteristics of a Cloud Based Internet of Things. In *Proc. Int. Conf. on Collaboration Technologies and Systems*, 2012.

[45] P. Langendoerfer, K. Piotrowski, M. Diaz, and B. Rubio. Distributed Shared Memory as an Approach for Integrating WSNs and Cloud Com-

puting. In *Proc. 5th Int. Conf. on New Technologies, Mobility and Security*, 2012.

[46] Cosm. `cosm.com`.

[47] João Pedro Sousa and David Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, WICSA 3, pages 29–43, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[48] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, pages 74–83, October 2002.

[49] E. Aitenbichler, J. Kangasharju, and M. Mühlhäuser. MundoCore: A light-weight infrastructure for pervasive computing. *Pervasive and Mobile Computing*, pages 332–361, 2007.

[50] Pankesh Patel, Animesh Pathak, Damien Cassou, and Valérie Issarny. Enabling high-level application development in the Internet of Things. In *Proceedings of the 4th International Conference on Sensor Systems and Software*, 2013.

[51] Sumi Helal, William Mann, Hicham El-Zabadani, Jeffrey King, Youssef Kaddoura, and Erwin Jansen. The gator tech smart house: A programmable pervasive space. *Computer*, pages 50–60, March 2005.

[52] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A. J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An operating system for the home. In *Proc. 9th USENIX Conf. on Networked Systems Design and Implementation*, 2012.

[53] SenseAware powered by FedEx. `goo.gl/zKc3Q`.

[54] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE*, pages 177–186, 1998.

[55] Richard N. Taylor, Nenad Medvidovic, and Peyman Oreizy. Architectural styles for runtime software adaptation. In *WICSA/ECSA*, pages 171–180, 2009.

[56] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *30th international conference on Software engineering*, pages 899–910, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1370175.1370181.

[57] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[58] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv. – to appear*.

[59] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[60] Elisabetta DiNitto, Carlo Ghezzi, Andreas Metzger, Mike P. Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3-4):313–341, 2008.

[61] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-538-3. doi: http://doi.acm.org/10.1145/1562112.1562118.

[62] Pascal Costanza. Language constructs for context-oriented programming. In *In Proceedings of the Dynamic Languages Symposium*, pages 1–10. ACM Press, 2005.

[63] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. Context oriented programming in highly concurrent systems. In *COP '10: International Workshop on Context-Oriented Programming, co-located with ECOOP 2010*, Maribor, Slovenia, 2010, (to appear).

[64] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. pages 2–28. Springer-Verlag, 2003.

[65] Philip Greenwood and Lynne Blair. L.: Using dynamic aspect-oriented programming to implement an autonomic system. Technical report, Proceedings of the 2003 Dynamic Aspect Workshop (DAW04 2003), RIACS.

[66] William E. Byrd. Web programming with continuations. Technical report, Unpublished Tech. Report, available at http://double.co.nz/pdf/continuations.pdf, 2002.

[67] Paul Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 211–, Washington, DC, USA, 2001. IEEE Computer Society. URL `http://portal.acm.org/citation.cfm?id=872023.872573`.

[68] Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 23–33, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: http://doi.acm.org/10.1145/351240.351243. URL `http://doi.acm.org/10.1145/351240.351243`.

[69] Luca Mottola, Thiemo Voigt, Fredrik Österlind, Joakim Eriksson, Luciano Baresi, and Carlo Ghezzi. Anquiro: enabling efficient static verification of sensor network software. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, SESENA '10, pages 32–37, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-

969-5. doi: http://doi.acm.org/10.1145/1809111.1809122. URL http://doi.acm.org/10.1145/1809111.1809122.

[70] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 267–276, New York, NY, USA, 2003. ACM. ISBN 1-58113-743-5. doi: http://doi.acm.org/10.1145/940071.940107. URL http://doi.acm.org/10.1145/940071.940107.

[71] Peng Li and John Regehr. T-check: bug finding for sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 174–185, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-988-6. doi: http://doi.acm.org/10.1145/1791212.1791234. URL http://doi.acm.org/10.1145/1791212.1791234.

[72] Sinem Coleri, Mustafa Ergen, and T. John Koo. Lifetime analysis of a sensor network with hybrid automata modelling. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, WSNA '02, pages 98–104, New York, NY, USA, 2002. ACM. ISBN 1-58113-589-0. doi: http://doi.acm.org/10.1145/570738.570752. URL http://doi.acm.org/10.1145/570738.570752.

[73] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.

[74] Marco Antoniotti and Aleks Göllü. Shift and smart-ahs: a language for hybrid system engineering modeling and simulation. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, DSL'97, pages 14–14, Berkeley, CA, USA, 1997. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267950.1267964.

[75] P.C. Olveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in real-time maude. *Parallel and Distributed Processing Symposium, International*, 0:157, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2006.1639414.

[76] PeterCsaba Ölveczky and José Meseguer. Specification and analysis of real-time systems using real-time maude. In Michel Wermelinger and Tiziana Margaria-Steffen, editors, *Fundamental Approaches to Software Engineering*, volume 2984 of *Lecture Notes in Computer Science*, pages 354–358. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21305-5. doi: 10.1007/978-3-540-24721-0_26. URL http://dx.doi.org/10.1007/978-3-540-24721-0_26.

[77] Marta Z. Kwiatkowska, Gethin Norman, and Jeremy Sproston. Probabilistic model checking of the ieee 802.11 wireless local area network protocol. In *PAPM-PROBMIV*, pages 169–187, 2002.

[78] Paolo Ballarini and Alice Miller. Model checking medium access control for sensor networks. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 255–262, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 978-0-7695-3071-0. doi: 10.1109/ISoLA.2006.16. URL http://portal.acm.org/citation.cfm?id=1396805.1397116.