# Building Internet of Things Software with ELIoT

Alessandro Sivieri[a], Luca Mottola[a,b], Gianpaolo Cugola[a]

[a]*Politecnico di Milano, Italy*
[b]*SICS Swedish ICT*

## Abstract

We present ELIoT, a development platform for Internet-connected smart devices. Unlike most solutions for the emerging "Internet of Things" (IoT), ELIoT allows programmers to implement functionality running *within* the networks of smart devices without necessarily leveraging the external Internet, and yet enables the integration of such functionality with Internet-wide services. ELIoT thus reconciles the demand for efficient localized performance, e.g., reduced latency for implementing control loops, with the need to integrate with the larger Internet. To this end, ELIoT's programming model provides IoT-specific inter-process communication facilities, while its virtual machine-based execution caters for the need of software reconfiguration and the devices' heterogeneity. Moreover, ELIoT addresses network-wide integration concerns by enabling standard-compliant interactions through REST and CoAP interfaces, with the added ability to dynamically reconfigure REST interfaces as application requirements evolve. We demonstrate the features and effectiveness of ELIoT based on a smart-home application, and quantitatively derive performance figures atop two hardware platforms compared to implementations using plain C or Java using the AllJoin framework. Compared to the C implementation, our results indicate that the performance cost for the increased programming productivity brought by ELIoT is still viable; for example, memory consumption in ELIoT is comparable, whereas the processing overhead remains within practical limits. Compared to the Java implementation using AllJoin, ELIoT provides a similar level of abstraction in programming, with much better performance both in memory consumption and processing overhead.

*Keywords:* Programming, Internet of Things

*Email addresses:* `alessandro.sivieri@polimi.it` (Alessandro Sivieri), `luca.mottola@polimi.it` (Luca Mottola), `gianpaolo.cugola@polimi.it` (Gianpaolo Cugola)

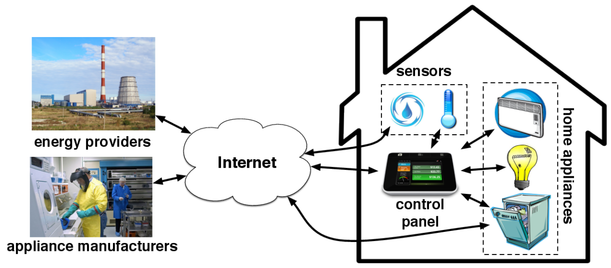


Figure 1: Smart-home application.

## 1. Introduction

The "Internet of Things" (IoT) is emerging from sensors and actuators aboard physical objects equipped with computing capabilities and able to access the larger Internet. Most often, a blend of *localized* and *Internet-wide* interactions characterizes IoT applications, as we exemplify next. How to effectively develop application software for such settings is an open problem [1]: currently available platforms rarely provide support for implementing applications that combine sharply different interaction patterns [2, 3, 4]. This greatly impacts the operational costs of IoT systems [5]. The same platforms often pay little tribute to the concerns arising when integrating different IoT systems. When standard-compliant interfaces are supported [6], they are typically carved in stone and therefore unable to accommodate evolving requirements.

**Problem.** Figure 1 describes an example smart-home [7] application. A control panel provides a user interface to coordinate the operation of several home appliances, such as HVAC systems, kitchen machines, and in-house entertainment, possibly based on environmental conditions gathered through sensors. Users input to the control panel their preferences, *e.g.*, the desired average temperature, and constraints, *e.g.*, the latest time for a dishwasher to complete washing.

Based on this information, per-appliance models of expected energy consumption, and energy prices found on the Internet, the control panel determines a schedule of activities to meet the user preferences while minimizing energy consumption, *e.g.*, by operating the dishwasher when energy is cheapest, but within the user constraints. Meanwhile, the control panel offers information on the instantaneous energy consumption over the Internet. The energy provider uses this information to estimate the city-wide load and to take informed decisions in case of unexpected peaks. Individual appliances should also be reachable through the Internet, *e.g.*, for appliance manufacturers to update their on-board software.

In this application, *localized* interactions are required to efficiently realize the control loops to configure home appliances based on user preferences and sensed data. On the other hand, *Internet-wide* interactions characterize the exchange of information between the smart-home installation and energy providers or appliance manufacturers. These traits are germane to many IoT applications [1, 8], including patient monitoring [9], vehicular traffic control [10] and smart logistics [11].

Although the devices typically employed in this kind of IoT applications feature sufficient resources to implement localized interactions [12, 13], existing software platforms almost exclusively delegate the application-specific functionality to the Internet, *e.g.*, using Cloud services such as Xively [14], ThingSpeak [15], and OpenSense [16]. There, sensor data is processed and actuator commands are remotely generated. The application logic thus resides entirely outside the networks of smart devices. This approach provides a quick path to working implementations, but it falls short if stricter performance requirements, *e.g.*, low latency for closed-loop control, become mandatory.

**Contribution and road-map.** This paper presents ELIoT, a programming platform for Internet-connected smart devices, which allows programmers to implement functionality running *within* the local network, while still supporting interactions with Internet-wide services. ELIoT is based on three cornerstones:

1) An IoT-tailored programming model: this includes, for example, dedicated language constructs to discern different communication guarantees due to the unreliability of the wireless channel, and dedicated addressing schemes to effectively support IoT interactions.
2) Support for standard-compliant interactions through REST and CoAP interfaces: while the latter seamlessly enable embedding low-power sensors and actuators in ELIoT applications, we also allow dynamic reconfiguration of REST interfaces to keep up with evolving requirements.
3) A custom run-time system fitting embedded devices the size of a gum stick and costing less than 10$: this also includes integrated simulation support for testing and debugging, with the ability of running hybrid scenarios that include simulated and real devices.

ELIoT programs are written in a dialect of Erlang [17]: an industry-strength language originally designed for fault-tolerant applications in the telecommunication domain. Erlang provides a stepping stone to implement IoT applications, because of its support for parallel and distributed programming.

Our evaluation indicates that ELIoT allows programmers to obtain concise code that is easy to debug, maintain, and reason about. The corresponding performance penalty is limited: by assessing the performance of a fault-tolerant ELIoT implementation of the smart-home application against a C-based counterpart with no embedded fault tolerance, we show that the overall memory consumption is comparable to the C implementation, whereas CPU usage remains within practical limits. Such a C-based implementation represents current practice in embedded system programming [18]. Nevertheless, compared to a non-fault tolerant Java implementation using the AllJoin framework, ELIoT shows orders of magnitude smaller memory consumption and CPU overhead. Notably, AllJoin explicitly targets IoT applications with requirements akin to ours, while Java offers similar levels of abstraction as ELIoT and a virtual machine-based implementation as well.

The paper unfolds as follows. In Section 2, we further analyze the smart-home application, which serves as a running example throughout the paper. Section 3 provides a concise Erlang primer. We describe ELIoT's programming model in Section 4. The support to standard-compliant interfaces, along with their dynamic reconfiguration, is discussed in Section 5, whereas ELIoT's run-time system is illustrated in Section 6. Next, Section 7 reports on our experimental evaluation. We end the paper by surveying related work in Section 8 and with concluding remarks in Section 9.

## 2. Motivating Application

The smart-home scenario we hint in the Introduction provides a paradigmatic example of the issues in developing IoT applications. Here we discuss a base design for this application, together with different deployment scenarios.

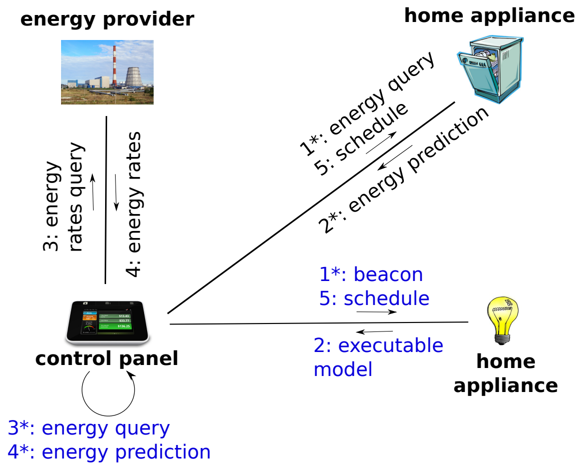
**Base design.** The devices in Figure 1 generally demand Internet access, *e.g.*, the control panel must be able to obtain energy rates from the provider and be accessible from the Internet, while appliance manufacturers must be able to remotely update the appliances' on-board software. At the same time, a *local* control loop, guided by the control panel, is beneficial to reduce communication costs and improve performance. In particular, the control panel acts as a front end for the users and coordinates the appliances' activities, dealing with:

***Functionality F1:*** discovery and bookkeeping of home appliances, obtaining the data to compute their operating schedules;

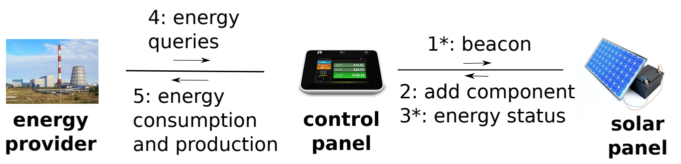
***Functionality F2:*** processing of the user inputs and computation of a schedule of appliance operation;

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

To ease the installation, smart-home devices are expected to feature wireless communication. Because of this, one designs the discovery functionality required in **F1** using a soft-state approach [19]. 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 an acknowledgment, the control panel removes the appliance from the application state.



(a) Scenario A and B.



(b) Scenario C.

Figure 2: Different scenarios in the smart-home application.

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

***Scenario A:*** if home appliances can locally compute their expected energy consumption, one can design the schedule computation of **F2** by issuing remote queries from the control panel to obtain the necessary information. This is shown in the black sequence of message exchanges in Figure 2a: 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 (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).

***Scenario B:*** if an appliance is computationally 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 at the control panel. The blue sequence of message exchanges in Figure 2a illustrates the

```
% Simple function returning the double of its input
double(Number) -> 2 * Number.

% Receive messages, process them, and return results to the original sender
loop() ->
    % Extract the first message from the queue (blocking)
    receive
        % Pattern match the content of the message
        {msg_type_1, SenderPID, ListOfNumbers} ->
            % Apply function Double to the whole list, element by element
            Result = lists:map(Double, ListOfNumbers),
            % Send the result back to the original sender
            SenderPID ! Result;
        % A different content for the message
        {msg_type_2, SenderPID, Content} ->
        [...]
    end,
    % Recursive call to parse next message in queue (or wait for a new message)
    loop()
end.
```

Figure 3: Erlang code sample.

corresponding interactions, which require computationally-constrained appliances to provide the control panel with an executable model of their expected energy consumption. The light fixture acknowledges the control panel's beacon (step 1) by shipping the model to compute its energy consumption (step 2). The control panel locally runs the model (step 3) to compute an estimate of the fixtures' energy consumption (step 4) before determining and transmitting the schedule (step 5).

***Scenario C:*** if some devices run different platforms, the necessary coordination must rely on standard-compliant interfaces. Such interfaces may serve to access low-power sensors and actuators, but they 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 2b, whenever this happens, the control panel should offer an additional interface to query the amount of produced energy. 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-compliant and vendor-independent manner, *e.g.*, using a REST interface.

## 3. Erlang Primer

ELIoT devices are programmed using a dialect of Erlang: an industrial-strength functional language designed to ease development of communication protocols, data manipulation algorithms, and distributed applications.

Erlang's concurrency model follows the actor model [20]: Erlang processes are named entities that do not share data, but communicate through asynchronous message passing, only. The example code in Figure 3 shows the core of an Erlang process that waits for incoming messages, processes them, and returns the result to the original sender. The **`receive`** statement inline 7 takes the first message from the process' incoming queue, while the **`!`** operator is used at line 13 to communicate the result back to the original sender. 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.

As shown at line 9 and 15, distinguishing between message types is specified declaratively using *pattern matching*, i.e., by stating constraints on the message format. In our example, **`msg_type_1`** and **`msg_type_2`** are two atoms that appear at the beginning of messages to distinguish them, while **`SenderPID`**, **`ListOfNumbers`**, and **`Content`** are unbound variables that are assigned a value at the time of performing the pattern matching. The same mechanism also allows one to parse and filter binary data, such as message payloads, using very compact code, as shown later in the paper. This is an asset for implementing low-level communication protocols, as often required in IoT applications.

Erlang code is compiled into a bytecode, which is interpreted or compiled just-in-time by a virtual machine (VM). ELIoT borrows the same approach, which elegantly addresses the issue of hardware heterogeneity typical of IoT applications. However, 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. Finally, debugging and testing IoT applications cannot be oblivious to the real-world interactions IoT systems are exposed to. ELIoT tackles these issues as described next.

## 4. Communication and Coordination in ELIoT

ELIoT's dedicated language constructs concerns four key aspects of IoT inter-process communication and coordination: *i)* handling different communication guarantees, *ii)* supporting code migration and remote process spawning, *iii)* offering extended addressing schemes, and *iv)* providing access to low-level information from the networking stack.

**Running example.** To make our explanation concrete, we consider the smart-home application introduced above. Figure 4 reports snippets of ELIoT code that

```
 1  % Define some char constants (1 byte) used as message headers, plus the timer for beacons
 2  -define(BCON, $M).
 3  -define(APPLIANCE, $A).
 4  -define(APPLIANCE_LOCAL, $L).
 5  -define(TIMER, 60000).
 6
 7  % Define the 'appliance' record with 3 fields: the appliance's IP address, plus
 8  % the process id and list of parameters of the appliance's consumption model
 9  -record(appliance, {ip, pid = none, parameters = []}).
10
11  % Main function to handle incoming messages; it takes the set of known appliances (a dictionary)
12  receiver(Appliances) ->
13    receive
14      [...]
15      % On receiving the timer self message...
16      timer ->
17        % Unreliably broadcast a beacon to processes named 'appliance'
18        Msg = <<?BCON:8>>,
19        {appliance, all} ~ Msg,
20        % Re-send the timer self-message to myself, after TIMER ms
21        erlang:send_after(?TIMER, self(), timer),
22        % Recurse to parse next message
23        receiver(Appliances);
24      % On receiving a message coming from a neighboring appliance...
25      {RSSI, SourceAddress, Content} ->
26        case Content of
27          % If first byte equals APPLIANCE, the consumption model runs remotely
28          <<?APPLIANCE:8, SerializedParameters/binary>>  ->
29            Pars = data:decode_params(SerializedParameters),
30            % Build a new set of appliances holding the new appliance information
31            NewRecord = #appliance{ip=SourceAddress, parameters=Pars},
32            NewAppliances = dict:store(SourceAddress, NewRecord, Appliances),
33            % Recurse to parse next message, passing the new set of appliances
34            receiver(NewAppliances);
35          % If first byte equals APPLIANCE_LOCAL, the consumption model runs locally
36          <<?APPLIANCE_LOCAL:8, Hash:20/binary, Len:8, SerializedName:Len/binary,
37            Code/binary>> ->
38            Name = erlang:binary_to_list(SerializedName),
39            % Spawn a new process to execute the received code
40            {Pid, Pars} = supervisor:start_model(Name, Code, Hash),
41            % Build a new set of appliances holding the new appliance information
42            NewRecord = #appliance{ip=SourceAddress, pid=Pid, parameters=Pars},
43            NewAppliances = dict:store(SourceAddress, NewRecord, Appliances),
44            % Recurse to parse next message, passing the new set of appliances
45            receiver(NewAppliances)
46        end
47    end.
```

Figure 4: Excerpt of control panel code.

implements different control panel's functionality: discovery of home appliances, as per functionality **F1** in the application base design (lines 16 to 23); gathering of the appliances' operating parameters, as per scenario **A** (lines 28 to 34); and installing of the executable model of an appliance's expected energy consumption, as per scenario **B** (lines 36 to 45).

After defining constants and structured types, the code in Figure 4 defines the recursive function **`receiver`** run by the control panel (line 12), which takes the current set of known appliances as input. Processing suspends at the **`receive`** statement (line 13) and then unfolds depending on the type of received message.

**Communication guarantees.** As mentioned in Section 3, 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[1]. This is a strong assumption in the IoT scenarios we target, where wireless communication is the rule more than the exception. 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 implements unreliable, best effort, sending of messages. We see it at work in line 19 of Figure 4: after creating the single byte beacon (line 18), the control panel sends it unreliably using the **∼** operator.

Besides adding the **∼** operator, ELIoT also changes the semantics of the **`!`** operator. Instead of assuming reliable links and failing silently in presence of unrecoverable faults, it places a special **`nack`** message into the sender's incoming message queue whenever a communication fault happens that cannot be automatically recovered by the VM. This enables programmers to implement their own application-specific failure-handling mechanisms, possibly based on the actual destination and payload of the failing massage, which are returned as part of the **`nack`** message.

ELIoT saves memory and processing overhead for processes that do not require network interactions by adopting a two step approach in mapping processes to names. A process **`register`**s under a symbolic name to allow (local) communication without the hassle of knowing the process identifier assigned by the VM. For the process to become accessible from the network, its name must be explicitly **`export`**ed. It is this step that activates the (sometime expensive) run-time infrastructure that allows the process to be reached from remote devices.

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. Providing a best-effort message send operator, alongside a more reliable one, while explicitly requiring processes to be exported 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 of Erlang by allowing both message sending operators to be used with local processes, also. In this case, both operators straightforwardly guarantee message delivery.

**Code transfer and remote process spawning.** As we mentioned in Section 3, ELIoT uses a VM to execute a platform-independent bytecode. While elegantly

[1]Mainstream Erlang implementations use TCP to provide this guarantee.

supporting the heterogeneity typical of IoT scenarios, this approach also allows code fragments to be sent over the network from device to device. This, together with the ability of dynamically spawning processes across devices, eases the dynamic (re)deployment of distributed applications. Devices can be dynamically added new capabilities by transferring the code that implements them and dynamically spawning the processes that execute such code.

This feature is used at lines 36 to 40 of Figure 4, which implement scenario **B** of our running example. Such fragment of code parses messages containing a binary blob (line 36), links the received code to the application, and instantiates a process to run it under the control of a supervisor process (line 40). In this case the spawning of the new process is triggered by the device that receives the code, but in principle a remote device may also send some code to a remote device and remotely spawn the corresponding process. The fact that spawning a process remotely uses the same primitives as in a local setting, while the message-passing functionality remains the same for local or remote communication, also allows ELIoT applications to move functionality from a local context to a distributed setting with minimal effort.

**Addressing schemes.** The `!` operator, originally offered by Erlang, allows single processes to be easily reached once programmers know their unique identifier or the name they registered to, together with the address of the VM they run on. While intuitive and easy to use, this form of unicast communication is insufficient to efficiently support scenarios where a process needs to send a message to multiple other 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 protocols.

ELIoT supports these scenarios by offering a richer addressing scheme than Erlang. In particular, ELIoT messages addressed to {`n, all`} arrive at processes registered under name `n` running on all reachable VMs[2]. We use this feature to implement the discovery of new appliances in Figure 4 (line 19). The same addressing scheme may be used within the `spawn` primitive, e.g., when a new functionality is to be deployed on multiple nodes at once. To further control the nodes where process spawning must happen, programmers may use ad-hoc *scoping filters*. They express a condition—in the form of a lambda function—that predicates over the devices' environment variables or that invokes functions available within the application itself. The process is actually spawned only onto those nodes where the scoping filter evaluates true.

[2]The ELIoT prototype implements the sending to `all` by using broadcast UDP; thus, the span of message spreading (and the notion of reachability) depends on the network configuration.

**Low-level network stack information.** Full isolation of the various layers that build a networking stack is sometimes impossible to achieve and often not beneficial. Some form of cross-layering is often required to improve efficiency, especially with embedded devices and wireless communication, which are the norm for IoT.

ELIoT makes these considerations concrete by exposing information coming from the networking stack to the receiver. More specifically, while Erlang fills the incoming message queue of the receiver only with the payload of the message, ELIoT's *communication driver* explicitly exposes additional information. In the current prototype, the IP address of the source node and the Received Signal Strength Indicator (RSSI) obtained from the radio are added, but the communication driver can be extended to add other information. Line 25 of Figure 4 shows how this information is easily accessible. This sharply contrasts the way programmers access similar information using low-level embedded system languages, like C. 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 [21] required for location-aware services.

## 5. Standard-compliant Interfaces

IoT applications are foreseen to emerge from the integration of a plethora of different platforms communicating through standard-compliant interfaces [22].

One such example is the Constrained Application Protocol (CoAP) [23]: an IETF proposal to allow wireless sensors and actuators to collaborate over low-power lossy networks. To accommodate for similarly constrained devices, we implement the CoAP standard in ELIoT and integrate it with the underlying run-time system. This allows an ELIoT node to natively integrate in a CoAP network, by invoking CoAP services to query a sensor or to send a command to an actuator. As an example, in our smart-home application, native support to CoAP may allow the control panel to query a CoAP-compliant weather station to enrich the scheduling algorithm with information about external conditions, or to directly control a CoAP-compliant appliance.

Dually, scenario **C** in the smart-home application requires standard-compliant access to ELIoT devices from an external entity. To this end, ELIoT provides support to *reconfigurable* REST interfaces, which provides two key features:

1) by facilitating the implementation of flexible REST interfaces, ELIoT enables rapid prototyping of distributed interactions based on standard protocols and

inter-operable message formats. For example, any web browser may be used to query sensors attached to an ELIoT node, with no ad-hoc programming.

2) ELIoT offers a means to *dynamically* extend existing REST interfaces. For example, upon installation of the solar panel of scenario **C**, the attached ELIoT device can deploy an additional function onto the control panel to extend its REST interface with a new operation that allows interested parties to access information on generated energy. The energy provider can access such data in a platform-independent manner, facilitating interoperability. Note that this kind of dynamic reconfiguration, enabled by ELIoT's ability to spawn new processes at run-time based on binary code received from the network, is rarely available in existing REST-enabled IoT platforms [6].

## 6. Run-time System

ELIoT provides two system functionality to effectively support the application execution and development: a lightweight VM that implements the language and a dedicated simulator for testing and debugging.

**Virtual machine.** Over time, Erlang has grown to support a wide range of scenarios, by means of a large set of libraries and a complex run-time infrastructure. Most of these features find limited application in IoT applications, unnecessarily increasing the hardware requirements. To address this issue, we develop a custom VM for ELIoT, which uses the Erlang VM as a foundation, keeping the available functionality to the bare minimum required in our target applications, and integrating the communication and coordination extensions that are unique to ELIoT.

At the communication layer, the ELIoT VM uses a custom networking stack with a double objective: improving efficiency and supporting the new communication primitives and addressing mechanisms described in Section 4. In particular, we employ UDP-based communication instead of TCP[3]. This applies to support both the reliable and unreliable communication primitives, and for remote spawning of processes. On top of UDP we implemented our own reliability layer, which supports the **`nack`** mechanisms described in Section 4. For better efficiency we also simplify the whole communication layer, limiting it to the minimum functionality required for the IoT scenarios we target.

As a result of this work the ELIoT VM has very low hardware requirements, especially in terms of memory consumption. This enables ELIoT to run on devices that are quite unusual in the traditional Erlang realm. We test two such platforms: *i)*

[3]In general, this is a custom choice, which can be easily changed by providing a different implementation for the ELIoT's communication driver.

a Raspberry Pi board model A with 256 MB of RAM, and *ii)* a custom embedded board with a RT3050 MIPS processor called "Carambola", featuring 32 MB of RAM and 8 MB of embedded flash. Both can run ELIoT.

**Simulator.** Debugging and testing IoT applications is a key area scarcely supported by most platforms. Gaining the required visibility into the system state, in particular, is deemed to be a crucial issue [24]. By leveraging ELIoT's VM-based run-time and the blurred distinction between local and distributed functionality, we develop a custom simulator that allows programmers:

- to simulate an entire system by instantiating a set of virtual nodes running *unmodified* ELIoT code;
- to model communication between nodes according to *real* wireless traces for increased fidelity[4];
- to *interact* with the simulation, if required, via a shell, e.g., to proactively inject messages or to overhead transmitted ones;
- to run a *hybrid deployment* where virtual nodes seamlessly interact with physical devices[5], thus creating a hardware-in-the-loop configuration [26].

ELIoT programmers can thus start debugging a system in a fully simulated deployment, and then progressively move to a setting where the execution also spans physical nodes. This retains visibility into the system state through the simulated nodes, but it also allows one to check the execution on real devices and the interactions with the physical environment. As we discuss next, we leverage ELIoT's simulator for debugging and testing our implementation of the smart-home application, using a Raspberry Pi as the control panel and simulated nodes as home appliances. This happens with the guarantee that the code being tested coincides, line by line, with the code that developers deploy.

## 7. Evaluation

We evaluate ELIoT by considering two aspects: the benefits it brings to developers' productivity and the run-time overhead it introduces to offer such benefits.

As a baseline for comparison, we use a C implementation of the smart-home application that realizes the same core functionality using the *pthread* library for multi-processing and standard UDP sockets for communication. This largely reflects the current practice in programming networked embedded systems [18]. To

[4]We use the traces from the TOSSIM simulator [25]. Using different traces is possible by developing the needed model translation.

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

provide a comparison with a platform expressly conceived for IoT development, we consider the AllJoyn [27] framework, using the Java language. AllJoyn is a state-of-the art, open-source networking framework developed by a consortium that includes most of the key players in the IoT panorama. The goal of the framework is to provide as easy-to-use platform to address the communication needs of IoT devices, covering, like ELIoT, both local and remote interactions. AllJoyn supports multiple platforms (Android, iOS, Linux, Open WRT, OS X, and Windows), languages (mainly C++, ObjectiveC, and Java), and networking technologies (Bluetooth and WiFi). Our decision of using Java is motivated by the desire to compare a programming environment whose ease-of-use, level of abstraction, and VM-based implementation are akin to ELIoT.

## 7.1. Benefits to IoT Software Development

ELIoT provides two benefits to programmers: it increases their productivity by rising the level of abstraction compared to low-level languages, and it eases debugging with custom tools.

**Programmers' productivity.** It is notoriously difficult to objectively compare the implementation effort using different programming languages. In absence of a precise tool, measuring the lines of code provides a rough, yet quantitative indication often used in the literature [28]. 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. The AllJoyn-based implementation requires 1408 lines of code. The latter shows better figures than C but still more than double the size of the ELIoT version.

These improvements become even more relevant as one considers that the C and AllJoyn implementations only provide 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 application supervisor, which handles process crashes as well as the testing and debugging services. These functionality are not available in the C and AllJoyn implementations. 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 compared to the C implementation and a 67% reduction compared to the the AllJoyn implementation. Such a big difference makes the result, even in presence of an approximate metric like the number of code lines, hardly questionable.

Beyond the raw numbers, the higher level of abstraction in ELIoT improves code readability, facilitating reuse and maintenance. This becomes visible by looking at the *structure* of the control panel code, shown in Figure 4. 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 declaratively by pattern matching.

As an example, line 36 in Figure 4 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 bytecode[6]. The name, hash, and code of the received function are then passed to the application supervisor (line 40) to spawn a new process executing the code and to monitor its execution should run-time errors occur.

Figure 5 provides additional insights into the expressive power of ELIoT, focusing on deserializing the operating parameters of a newly discovered appliance, as required in line 29 of Figure 4. In C, as shown in Figure 5a, 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, again using binary pattern matching. The **`decode_params`** function in line 3 of Figure 5b takes the message payload as input and invokes a function with the same name and an additional argument: an initially empty list of appliance 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, as in lines 11-12. Each parameter includes the length of the parameter's name (**`L1`**) followed by the name 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 a record prepended to the list of decoded parameters in the recursive call of line 17. Overall, the 25 lines of C code in Figure 5a reduce to 7 lines of (uncommented) ELIoT code in Figure 5b.

A comparison with the AllJoyn implementation is harder as AllJoyn adopts an RPC-based communication model rather than a message-based one as in ELIoT and C. This choice has the benefit of hiding most of the communication details and in particular the steps required to serialize and de-serialize RPC parameters. At the same time, the need of supporting multiple languages and platforms requires AllJoyn programmers to use ad-hoc code to let AllJoin know the format and size of involved data types. Considering the data types that encode the operating parame-

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

```c
int deserialize_params(char *buf, GList **params) {
  unsigned int params_len;
  int tot, i;
  parameter_t *param = NULL;
  memcpy(&params_len, buf, sizeof(unsigned int));
  for (i = 0, tot = 0; i < params_len; ++i) {
    tot += deserialize_parameter(buf + sizeof(unsigned int) + tot, &param);
    *params = g_list_append(*params, (void *) param);
  }
  return sizeof(unsigned int) + tot;
}
int deserialize_parameter(char *buf,
                          parameter_t **param) {
  unsigned long name_len;
  parameter_t *p = NULL;
  p = malloc(sizeof(parameter_t));
  memset(p, 0, sizeof(parameter_t));
  memcpy(&name_len, buf, sizeof(unsigned long));
  p->name = g_string_new_len(buf + sizeof(unsigned long), name_len);
  memcpy(&p->type, buf + sizeof(unsigned long) +  name_len, 1);
  memcpy(&p->value, buf + sizeof(unsigned long) + name_len + 1, sizeof(uint8_t));
  memcpy(&p->ro, buf + sizeof(unsigned long) + name_len + 1 + sizeof(uint8_t), sizeof(uint8_t));
  *param = p;
  return sizeof(unsigned long) + name_len + 1 + 2*sizeof(uint8_t);
}
```

(a) C implementation.

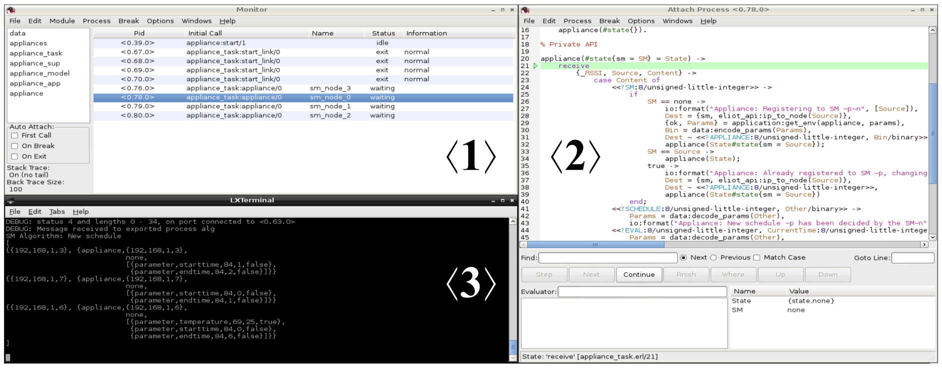
```erlang
% Decode Payload by calling the two-args version of the function passing an empty list,
% which will be filled with the data extracted from the payload
decode_params(Payload) -> decode_params(Payload, []).

% Pattern matching on the first arg: if the binary variable is empty, then we finished
% (we reached the base case for the recursion) and we can return the ListOfPars...
decode_params(<<>>, ListOfPars) -> ListOfPars;
% ... otherwise, the first byte (L1) contains the length of the parameter's name (next field),
% and the following bytes represent: its type, its value, and it being read-only; the rest
% of the payload contains other parameters that will be extracted in the next (recursive) call
decode_params(<<L1:8, SerializedName:L1/binary, Type:8/unsigned-integer,
    Value:8/unsigned-integer, Ro:8/unsigned-integer, Rest/binary>>, ListOfPars) ->
    % Fill a new record with the extracted content
    NewRecord = #parameter{name = erlang:binary_to_list(SerializedName),
        type = Type, value = Value, ro = Ro},
    % Recursive call to continue parsing the payload. The new record is saved into the list
    decode_params(Rest, [NewRecord|ListOfPars]);
```

(b) ELIoT implementation.

Figure 5: Deserializing appliance operating parameters.

ters of a newly discovered appliance, the code programmers need to implement to let AllJoin correctly handle this information amounts to 12 Java lines. In addition, 70 Java lines are needed to implement the **hashCode** and **equals** methods that need to be redefined for Java objects passed around a network. This compares with the 7 lines of ELIoT code mentioned above.

One might argue that the compact ELIoT code, which results from its functional paradigm, may 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].


Monitor
File Edit Module Process Break Options Windows Help
⟨1⟩
Attach Process <0.78.0>
File Edit Process Break Options Windows Help
⟨2⟩
LXTerminal
File Edit Tabs Help
⟨3⟩
Find:
Next Previous Match Case
Goto Line:
Step Next Continue Finish Where Up Down
Evaluator:

data
appliances
appliance_task
appliance_sup
appliance_model
appliance_app
appliance

| Pid | Initial Call | Name | Status | Information |
|---|---|---|---|---|
| <0.39.0> | appliance:start/1 | | idle | |
| <0.67.0> | appliance_task:start_link/0 | | exit | normal |
| <0.68.0> | appliance_task:start_link/0 | | exit | normal |
| <0.69.0> | appliance_task:start_link/0 | | exit | normal |
| <0.70.0> | appliance_task:start_link/0 | | exit | normal |
| <0.76.0> | appliance_task:appliance/0 | sm_node_3 | waiting | |
| <0.78.0> | appliance_task:appliance/0 | sm_node_0 | waiting | |
| <0.79.0> | appliance_task:appliance/0 | sm_node_1 | waiting | |
| <0.80.0> | appliance_task:appliance/0 | sm_node_2 | waiting | |

Auto Attach:
☐ First Call
☐ On Break
☐ On Exit
Stack Trace:
On (no tail)
Back Trace Size:
100

```
    appliance(#state{}).

% Private API

appliance(#state{sm = SM} = State) ->
    receive
        {_RSSI, Source, Content} ->
            case Content of
                <<?SM:8/unsigned-little-integer>> ->
                    if
                        SM == none ->
                            io:format("Appliance: Registering to SM ~p~n", [Source]),
                            Dest = {sm, eliot_api:ip_to_node(Source)},
                            {ok, Params} = application:get_env(appliance, params),
                            Bin = data:encode_params(Params),
                            Dest ~ <<?APPLIANCE:8/unsigned-little-integer, Bin/binary>>
                            appliance(State#state{sm = Source});
                        SM == Source ->
                            appliance(State);
                        true ->
                            io:format("Appliance: Already registered to SM ~p, changing
                            Dest = {sm, eliot_api:ip_to_node(Source)},
                            Dest ~ <<?APPLIANCE:8/unsigned-little-integer>>,
                            appliance(State#state{sm = Source})
                    end;
                <<?SCHEDULE:8/unsigned-little-integer, Other/binary>> ->
                    Params = data:decode_params(Other),
                    io:format("Appliance: New schedule ~p has been decided by the SM~n"
                <<?EVAL:8/unsigned-little-integer, CurrentTime:8/unsigned-little-intege
                    Params = data:decode_params(Other),
```

| Name | Value |
|---|---|
| State | {state,none} |
| SM | none |

State: 'receive' [appliance_task.erl/21]

```
DEBUG: status 4 and lengths 0 - 34, on port connected to <0.63.0>
DEBUG: Message received to exported process alg
SM Algorithm: New schedule
[
{{192,168,1,3}, {appliance,{192,168,1,3},
                           none,
                           [{parameter,starttime,84,1,false},
                            {parameter,endtime,84,2,false}]}}
{{192,168,1,7}, {appliance,{192,168,1,7},
                           none,
                           [{parameter,starttime,84,0,false},
                            {parameter,endtime,84,1,false}]}}
{{192,168,1,6}, {appliance,{192,168,1,6},
                           none,
                           [{parameter,temperature,69,25,true},
                            {parameter,starttime,84,0,false},
                            {parameter,endtime,84,6,false}]}}
]
```

Figure 6: Simulator user interface.

ELIoT also simplifies implementing concurrent functionality, by virtue of its functional nature and system support to multi-threading. As an example, mutexes and condition variables, required in C (but also in Java) to synchronize concurrent threads, are unnecessary with ELIoT. Already in the relatively simple smart-home application, nonetheless, C and Java programmers (albeit the latter with the help of higher-level language constructs) heavily rely on these synchronization primitives to coordinate access to the shared list of appliances. ELIoT programmers can 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.

**Testing and debugging.** The real-world dynamics and the decentralized operation of IoT applications complicate testing and debugging. The ELIoT simulator helps deal with these tasks by providing monitoring and inspection tools for hybrid configurations of real and simulated nodes.

Figure 6 shows the simulator at work. When debugging the smart-home application, we use a real Raspberry Pi to run the control panel, plus four simulated appliances. Developers interact with the ELIoT simulator in three ways: *i)* a *process monitor* shows the ELIoT processes running on simulated nodes, identified according to their **`register`**-ed names; *ii)* a *code monitoring* tool enables inspection of the currently running code and allows to step through instructions and set breakpoints, as well as to manipulate the values of variables; *iii)* a *custom shell* allows developers to trigger specific executions, *e.g.*, the schedule computation on the Raspberry Pi. The simulator then shows how the appliances answer to the control panel through the process and code monitors. The shell allows one to automatize these operations by scripting sequences of test cases.

The ELIoT simulator offers functionality that are rarely available using mainstream programming platforms for networked embedded systems [19]. The VM-based execution, together with the actor model that simplifies inter-process com-

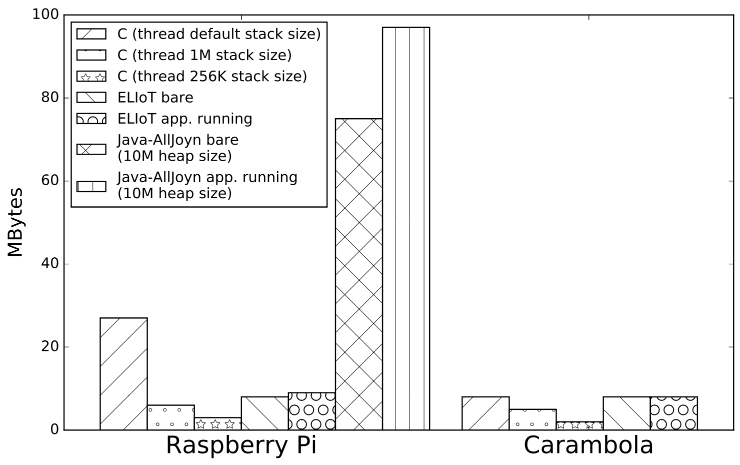


Figure 7: Memory consumption (pmap).

munications, facilitates building tools that effectively support developers in testing and debugging distributed functionality.

## 7.2. *System Performance*

Increasing developers' productivity 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, AllJoyn, and ELIoT implementations of the smart-home application by measuring memory consumption, CPU usage and power consumption, as well as network traffic and latency. For the C version, we perform this comparison on both embedded devices currently running the ELIoT VM, while the AllJoyn version is only tested on the Raspberry Pi board, since a Java VM is not available for the Carambola board.

**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 7 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 programmers with 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 programming, however, this is conducive to interesting observations: a naive C programmer who uses the default stack size[7] would build an application 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. 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. With the application running, instead, AllJoyn using Java consumes one order of magnitude more memory than ELIoT, even after fine tuning the Java VM's heap size.

To better characterize memory usage in AllJoyn and ELIoT, we separately assess the two VMs with no application loaded and when the smart-home application is running. As shown in Figure 7, it turns out that both VMs are responsible for most of the memory used. But while in ELIoT the application consumes a few additional KB, in AllJoyn using Java the amount of additional memory required to run the application is significant. This analysis points at the VM as an avenue for further improvements to battle the memory overhead in ELIoT. At the same time, it also suggests that the gap between C and the other two, higher-level platforms, would likely reduce with more complex applications, as the memory occupation due to the VM is a fixed cost that ELIoT and AllJoin pay once and for all, with ELIoT showing better relative performance on this metric.

**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 8 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 Caram-

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

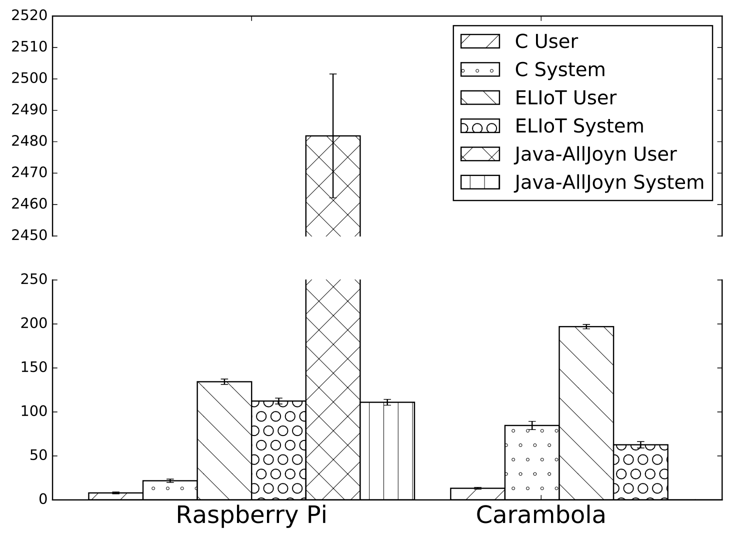


Figure 8: CPU times (in hundreds of seconds).

bola 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 latency 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. The numbers we gather from the AllJoyn implementation tell a different story. While the system time is comparable with ELIoT, the user time is 18 times greater. Considering that AllJoyn is a state-of-the-art platform for IoT development, this result puts ELIoT's performance in a different perspective. Albeit offering a high-level programming model similar to using Java with AllJoyn, ELIoT has a much lower overhead compared to a pure C implementation, providing a much better compromise between ease of use and performance.

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 9 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. The power consumption of the AllJoyn version reflects the CPU usage we reported above, consuming 30 times more than the ELIoT version.

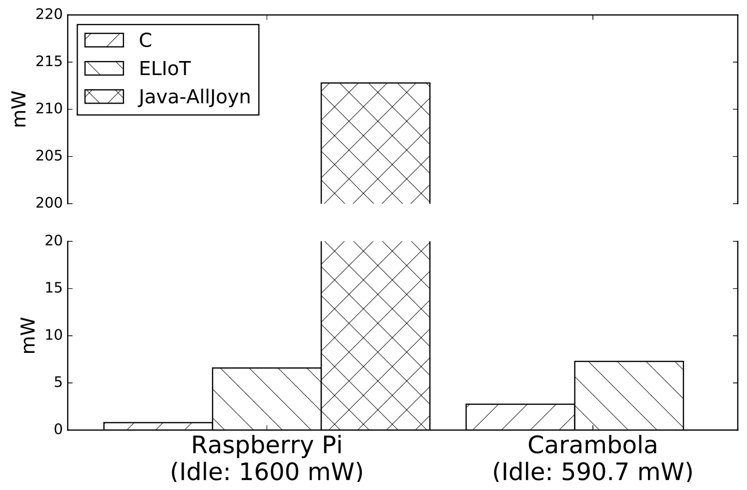


Figure 9: Power consumption. (The idle power consumption is factored out.)

On a general note, we may observe that 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.

**Network traffic and latency.** Using a standard network inspection tool, we measured the amount of bytes transferred through the network during a single iteration of the smart-home application. This includes several messages exchanged between the control panel and the appliances of our smart-home example. While the application payload of such messages in the same for the three platforms we tested, the header and format differ. For ELIoT this is a result of its specific features platform, such as the abstract addressing mechanism it provides, for example, to reach specific ELIoT processes within a given node. Using AllJoyn results in a complete change of the communication paradigm, which moves from message passing to RPC. In comparison with C, ELIoT shows a 10.21% overhead (2126 bytes vs. 1929). The *number* of messages, however, is the same in both implementations. Overall, this small overhead appears acceptable. Using AllJoyn results in much greater overhead, with a total traffic of 9572 bytes, adding to 496% overhead compared to C. The number of messages also greatly increases, as AllJoyn employs its own beaconing mechanism to discover new devices and maintain reachability and

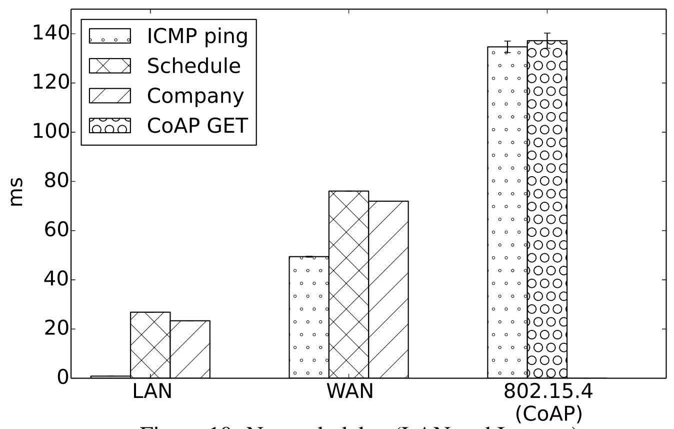


Figure 10: Network delay (LAN and Internet).

bookkeeping information for all devices involved.

We also measure the network latency of ELIoT messages in LAN and Internet-scale interactions. The first two sets of bars in Figure 10 show the round-trip time of two exemplary ELIoT messages used in our scenario: the "schedule" message exchanged between the control panel and an appliance to inform it of the new agreed schedule and the "company" message exchanged between the electrical company and the control panel. They are both representative examples of complex interactions that may happen in an ELIoT application.

The LAN measurements (first set of bars) have been taken using two Raspberry Pis on a wired connection to the same router, while the WAN measurements (second set of bars) have been taken using two Raspberry Pis situated in Italy and Sweden. The two devices execute a control panel and an appliance in the first case and a control panel and the company functionality in the second case. The chart shows also the network delay measured through ICMP ping messages. In all cases, the interactions have been executed 10000 times, and the plot shows the 95% confidence intervals, barely visible since they are under 0.5%. We observe 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 interface on the sender, traverse back the stack on the receiver, elaborate the response and send it back to the sender. We expect such small latency to be acceptable in most practical IoT applications.

To complement these measures, the rightmost bars in Figure 10 show the time required for the control panel running on a Raspberry Pi to invoke a CoAP service offered by a TMote Sky node running Contiki on a 802.15.4 network. As in the previous case, we also plotted the delay measured on the same 802.15.4 link using

ICMP v6 ping messages. We note that the time required to perform this interaction is below 150ms. More interesting is the comparison between the round trip time measured through ICMP and the CoAP invocation latency. The two measures looks very similar. This can be explained because of the Contiki-based implementation at the TMote Sky device, whereby there is not much difference in the processing to receive, decode, and answer an ICMP packet vs. a CoAP request. In both cases, most of the processing time is actually spent in the routing (RPL), MAC (802.15.4), and physical layers.

**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 where appliance manufacturers need to update the on-board software. Particularly, 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 goes from 50 ms on the Raspberry Pi to less than 20 ms on the Carambola: arguably acceptable in most practical IoT applications.

## 8. Related Work

Works closely related to ELIoT mainly target IoT software architectures and IoT application frameworks. From a conceptual standpoint, the body of work on sensor network programming and pervasive computing also shares some objectives with ELIoT, along with some existing application-specific frameworks.

**IoT architectures and frameworks.** Significant activities are undergoing to define software architectures for the IoT. At the lowest layers, for example, Calipso [32] aims to define a global network architecture for IPv6-based smart objects. The IoT6 project [22] exploits an IPv6-based network layer to build CoAP services atop. The IoT-A project [33] defines an architectural reference model for the interoperability of IoT devices, whereas Spitfire [34] investigates unified concepts for facilitating the effective development of IoT applications.

ELIoT is largely complementary to these efforts. It already integrates with the results of the IoT6 project thanks to the embedded support to CoAP, and fits the IoT-A architecture as a possible tool to implement the functionality offered by Internet-connected smart devices. Generally, 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 and perform communication and coordination across the network of devices and with Internet-wide services. ELIoT provides support for the latter aspects.

Integrating smart devices with the Internet may follow different approaches. Solutions exist to proactively export sensor data to the Internet, such as Publish/-Subscribe middleware [3] and shared memory systems [35]. Cloud-hosted platforms providing storage and processing facilities for sensor data also exist, such as Xively [14], ThingSpeak [15], and OpenSense [16]. Other solutions instead provide remote access to sensors and actuators from the Internet, such as sMAP [6]. A notion of "physical mashup" [36] is also emerging, *e.g.*, as in systems like IBM's Node-RED [37]. Mitton et al. [38] present a concept of sensor virtualization applied to a smart-city use case.

In all these approaches, the application logic runs outside the network of embedded sensor and actuators. This simplifies quickly prototyping IoT applications, yet it does not allow an efficient implementation of combined Internet-wide and localized interactions. ELIoT aims at efficiently enabling the latter by retaining the ability to coordinate with Internet-wide services. For example, as seen in the smart-home scenario, ELIoT developers can implement control loops that span neighboring ELIoT- enabled devices; coordinate with non ELIoT- enabled devices using standard protocols; and integrate them with externally-running services. Moreover, ELIoT's REST interface enables the integration with systems based on RESTful interactions, such as Actinium [4].

There also exist works tackling different facets of IoT applications. Srijan [39], for example, presents a model-driven development approach by establishing specific roles for the involved stakeholders, and by introducing domain-specific languages to model both the application and the underlying systems. Latronico et al. [40] present the notion of "accessor" as a way to tackle heterogeneity in IoT applications. An accessor is a software wrapper that exports the functionality of sensors, actuators, and Internet-wide services according to an actor-like model. These works are complementary to ELIoT, which focuses on providing effective programming and system support. For example, ELIoT may serve as a target language for Srijan, to simplify code generation. Even more strikingly, as ELIoT natively supports the actor model, it becomes a natural candidate for supporting the implementation and execution of accessors.

**Pervasive computing and sensor networking.** Most pervasive computing platforms are conceived as stand-alone systems, where Internet-wide interactions are typically mediated by ad-hoc gateways designed and implemented on a per-application basis. For example, Aura [41] and Gaia [42] focus on effective development of on-the-fly interactions between users and nearby pervasive computing devices, whereas MundoCore [43] provides a low-level framework and middleware for developing platforms integrating diverse devices, from mobile systems to mainstream PCs. Although MundoCore caters for effective integration of heterogeneous hardware—an issue we also tackle in ELIoT using a VM-based execution—these system do not tackle the problem of effectively developing systems featuring both Internet-wide and localized interactions.

Similar considerations apply to traditional sensor networking. Although based on a different hardware, existing solutions in the field [19] do enable the implementation of localized interactions—especially by deploying the application logic right onto the embedded devices—but lack support for Internet-wide interactions. Programming may occur at the operating system level [44, 45], by relying on custom virtual machines [46], or by using higher-level abstractions [19]. Conceptually, ELIoT aims at bringing the localized interactions already enabled by sensor network programming in Internet-connected embedded networks. For example, support to CoAP-based interactions in ELIoT helps achieve this goal.

**Application-specific frameworks.** We use a smart-home application to exemplify the use of ELIoT. Ad-hoc solutions exist for developing software in specific domains. For example, HomeOS [47] 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 inform business analysts at the back-end of item availability and market trends [11]. Such applications show similar combinations of localized and Internet-wide interactions as our smart-home example. ELIoT precisely aims at enabling both kinds of interactions within the same platform.

## 9. Conclusions

We presented ELIoT, a development platform for the IoT that allows developers to combine localized and Internet-wide interactions. ELIoT builds upon Erlang by adapting its inter-process communication facilities to the specifics of IoT applications, using custom language syntax and semantics. The VM-based execution supports the diverse IoT hardware and provides the necessary software reconfiguration capabilities. ELIoT nodes export reconfigurable REST interfaces for standard-compliant interactions, while a dedicated VM tailored to mainstream IoT devices supports the distributed executions of ELIoT applications, and a custom simulator aids testing and debugging using hybrid configurations of real and simulated devices.

By comparing, both qualitatively and quantitatively, the implementation of a smart-home application using ELIoT and standard C, we found that the former facilitates development by producing more concise and more readable code that is easier to test and debug. The performance penalty is, on the other hand, limited. For example, memory usage in ELIoT is often comparable to the C counterparts, whereas CPU usage remains within practical limits. The comparison with the state-of-the-art programming framework AllJoyn, paired with Java to offer a high-level programming style analogous to that of ELIoT, shows how the latter provides a much better compromise between ease-of-use and performance, with a much lower performance overhead.

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

## References

[1] F. Kawsar, G. Kortuem, B. Altakrouri, Supporting interaction with the internet of things across objects, time and space, in: Proc. Internet of Things Conf., 2010.
[2] Cosm, cosm.com.
[3] G. Fox, S. Kamburugamuve, R. Hartman, Architecture and Measured Characteristics of a Cloud Based Internet of Things, in: Proc. Int. Conf. on Collaboration Technologies and Systems, 2012.
[4] M. Kovatsch, M. Lanter, S. Duquennoy, Actinium: A RESTful runtime container for scriptable IoT applications, in: Proc. Int. Conf. on the Internet of Things, 2012.
[5] P. Marron, S. Karnouskos, D. Minder, A. Ollero, The Emerging Domain of Cooperating Objects, Springer, 2013.
[6] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, D. Culler, sMAP: a simple measurement and actuation profile for physical information, in: Proc. 8th ACM Conf. on Embedded Networked Sensor Systems, 2010.
[7] D. J. Cook, S. K. Das, How smart are our environments? An updated look at the state of the art, Pervasive Mob. Comput. 3 (2).
[8] D. Uckelmann, M. Harrison, F. Michahelles, Architecting the Internet of Things, Springer, 2011.
[9] K. Lorincz, B.-r. Chen, G. W. Challen, A. R. Chowdhury, S. Patel, P. Bonato, M. Welsh, Mercury: a wearable sensor network platform for high-fidelity motion analysis, in: Proc. 7th ACM Conf. on Embedded Networked Sensor Systems, 2009.
[10] R. Sen, A. Maurya, B. Raman, R. Mehta, R. Kalyanaraman, N. Vankadhara, S. Roy, P. Sharma, Kyun queue: a sensor network system to monitor road traffic queues, in: Proc. 10th ACM Conf. on Embedded Network Sensor Systems, 2012.
[11] SenseAware powered by FedEx, goo.gl/zKc3Q.
[12] BeagleBoard, beagleboard.org/Products/BeagleBone.
[13] Raspberry PI, www.raspberrypi.org.
[14] Xively, www.xively.com.
[15] ThingSpeak, www.thingspeak.com.
[16] OpenSense, open.sen.se.
[17] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007.
[18] M. Barr, A. Massa, Programming Embedded Systems, O'Relly Media, 2006.

[19] L. Mottola, G. P. Picco, Programming wireless sensor networks: Fundamental concepts and state of the art, ACM Compututing Surveys 43.

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

[21] K. Langendoen, N. Reijers, Distributed localization in wireless sensor networks: a quantitative comparison, Comput. Netw. 43 (4).

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

[23] Z. Shelby, K. Hartke, C. Bormann, Constrained application protocol (CoAP), `draft-ietf-core-coap-18` (Dec. 2013).

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

[25] P. Levis, N. Lee, M. Welsh, D. Culler, TOSSIM: accurate and scalable simulation of entire TinyOS applications, in: Proc. 1st ACM Conf. on Embedded Networked Sensor Systems, 2003.

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

[27] Alljoyn, `allseenalliance.org`.

[28] R. W. Sebesta, Concepts of Programming Languages, 9th Edition, Addison-Wesley Publishing Company, USA, 2009.

[29] U. Wiger, G. Ask, K. Boortz, World-class product certification using erlang, SIGPLAN Not. 37 (12).

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

[31] A. Carroll, G. Heiser, An analysis of power consumption in a smartphone, in: Proc. of the USENIX annual technical conference, 2010.

[32] CALIPSO: Connect All IP-based Smart Objects!, `www.ict-calipso.eu`.

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

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

[35] P. Langendoerfer, K. Piotrowski, M. Diaz, B. Rubio, Distributed Shared Memory as an Approach for Integrating WSNs and Cloud Computing, in: Proc. 5th Int. Conf. on New Technologies, Mobility and Security, 2012.

[36] D. Guinard, et al., A resource oriented architecture for the Web of Things, in: Internet of Things (IOT), 2010.

[37] IBM Node-RED, `www.nodered.org`.

[38] N. Mitton, et al., Combining Cloud and sensors in a smart city environment, Journal on Wireless Communications and Networking 2012 (1).

[39] P. Patel, A. Pathak, D. Cassou, V. Issarny, Enabling high-level application development in the Internet of Things, in: Proceedings of the 4th International Conference on Sensor Systems and Software, 2013.

[40] E. Latronico, E. Lee, M. Lohstroh, C. Shaver, A. Wasicek, M. Weber, A vision of swarmlets, Internet Computing, IEEE 19 (2) (2015) 20–28.

[41] J. a. P. Sousa, D. Garlan, Aura: an architectural framework for user mobility in ubiquitous computing environments, in: Proceedings of the IFIP 17th World Computer Congress, 2002.

[42] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, K. Nahrstedt, A middleware infrastructure for active spaces, IEEE Pervasive Computing 1.

[43] E. Aitenbichler, J. Kangasharju, M. Mühlhäuser, MundoCore: A light-weight infrastructure for pervasive computing, Pervasive and Mobile Computing 3.

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

[45] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, System architecture directions for networked sensors, SIGPLAN Not. 35 (11).

[46] N. Brouwers, K. Langendoen, P. Corke, Darjeling, A Feature-Rich VM for the Resource Poor, in: Proc. of the 7th ACM Conference on Embedded Networked Sensor Systems, 2009.

[47] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, P. Bahl, An operating system for the home, in: Proc. 9th USENIX Conf. on Networked Systems Design and Implementation, 2012.