ELIoT: Building Internet of Things Software Combining Localized and Internet-scale Interactions

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

[a]Politecnico di Milano, Italy
[b]Swedish Institute of Computer Science

Abstract

We present ELIOT, a development platform for Internet-connected smart devices. Unlike most solutions for the emerging "Internet of Things" (IoT), ELIOT allows to program functionality running *within* the networks of smart devices without necessarily leveraging the external Internet, and yet enables the integration of such functionality with Internet-side services. ELIOT thus reconciles the demand for efficient performance, e.g., minimum latency for implementing localized control loops, with the need to integrate with the larger Internet. ELIOT builds upon the Erlang language, adapted to the specifics of IoT programming with dedicated inter-process communication facilities. Its virtual machine (VM) based execution caters for the systems' heterogeneity and the software reconfiguration required in IoT scenarios. We demonstrate ELIOT based on a smart-home application, supporting Internet-scale interactions via REST interfaces that ELIOT provides in a reconfigurable manner. Our experimental results—obtained atop two hardware platforms against a C implementation of the smart-home core functionality—indicate that the performance cost for the increased programming productivity brought by ELIOT is still viable; for example, memory consumption in ELIOT is comparable to the C counterparts, and the processing overhead remains within practical limits.

Keywords: Programming, Internet of Things, Erlang

1. Introduction

Everyday's objects are increasingly equipped with computation and communication functionality; the latter providing the ability to exchange data with other

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

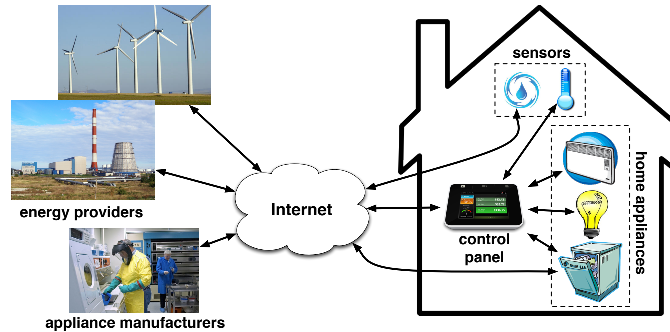

Figure 1: Smart-home application.

smart objects and to connect to the larger Internet. Sensors and actuators aboard these objects enable interactions with the real world. This combination creates an "Internet of Things" (IoT) enabling novel functionality and added value services [1]. Most often, a blend of *localized* and *Internet-scale* interactions characterize IoT applications, as we exemplify next. How to effectively develop application software for such settings is, however, an open problem [2].

Scenarios and problem. Figure 1 shows an example smart-home [3] 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. At the same time, the control panel offers information on the instantaneous energy consumption over the Internet. The energy provider leverages this information to estimate the overall city-wide load and to make informed decisions in case of unexpected peaks. Nevertheless, as for the control panel, also individual appliances should be reachable through the Internet, *e.g.*, for appliance manufacturers to update their on-board software.

As we further analyze in Section 2, in this application *localized* interactions are required to efficiently realize the control loops to set the operation of home appliances based on user preferences and sensed data. On the other hand, *Internet-scale* interactions characterize the information exchanges between the smart-home installation and energy providers or appliance manufacturers. This combination of traits is not unique to the smart home, and is germane to many current and

foreseeable IoT applications [2, 4]: examples range from patient monitoring [5] to vehicular traffic control [6] and logistics [7].

In these scenarios, although developers usually employ devices with sufficient resources to implement localized interactions without necessarily leveraging Internet-side services [8, 9], existing software platforms [10, 11] almost exclusively delegate the application-specific functionality to the Internet. There, sensor data is processed and actuator commands are remotely generated to be later re-injected in the embedded system. The application logic thus resides entirely outside the networks of smart devices. Although this provides a quick path to working prototypes, 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 system for Internet-connected smart devices, which allows to implement functionality running *within* the local network of embedded devices, while still supporting their integration with Internet-scale services. ELIoT is based on Erlang [12]: an industry-strength, functional programming language originally designed for fault-tolerant applications in the telecommunication domain, which we briefly introduce in Section 3. Erlang provides an ideal stepping stone to implement IoT applications, because of its advanced support for parallel and distributed programming and its VM-based run-time system.

As illustrated in Section 4, with ELIoT we adapt Erlang's programming model to the specifics of IoT applications and tailor the corresponding run-time system to the capabilities of typical IoT devices. This entails, for example, providing 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. At system level, as described in Section 5, ELIoT's design allows us to support reconfigurable REST interfaces, which provide inter-operability of ELIoT devices based on standard-compliant protocols. Nevertheless, unlike mainstream Erlang run-time systems, ELIoT runs on embedded devices the size of a gum stick and costing less than 40$. We also provide integrated simulation support for testing and debugging, with the ability of running hybrid scenarios that include simulated and real devices.

Our evaluation of ELIoT, reported in Section 6, indicates that ELIoT is beneficial for the development of IoT applications. It allows to obtain more concise code that is easier to debug, maintain, and reason about. The performance penalty to pay back such benefits 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 still comparable, whereas CPU usage is higher with ELIoT, but still within practical limits. Notably, the C-based implementation of the smart-home applica-

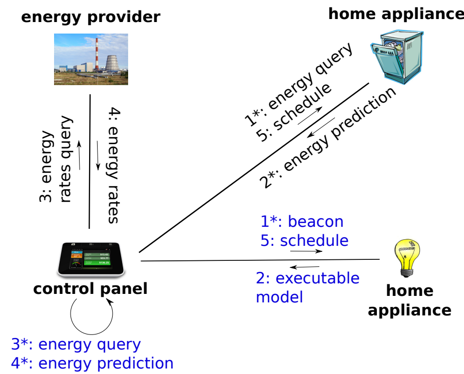

Figure 2: Scenario A and B.

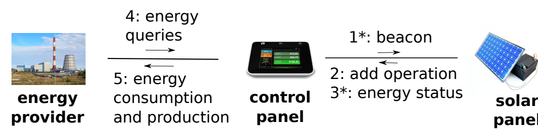

Figure 3: Scenario C.

tion arguably represents the current practice in programming networked embedded systems.

We end the paper by concisely surveying related work in Section 7 and by presenting our concluding remarks in Section 8.

[1]

2. Motivating Application

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

[1] **Luca: Was in the intro before, should be moved to the letter: "We described initial ideas about ELIoT in an earlier position paper [13]. Here we revisit, extend, concretely realize, and thoroughly evaluate these ideas against increasingly challenging application scenarios. Specifically, rather than exporting ELIoT's functionality as a library, we immerse them in Erlang's syntax by means of dedicated language constructs. We revisit Erlang's inter-process communication facilities to account for the unreliability of wireless communications, while enabling standard-compliant distributed interactions based on reconfigurable REST interfaces. Finally, we evaluate these features—both qualitatively and quantitatively—in a concrete application implementation against its C-based counterpart, which arguably represent the current practice in programming networked embedded systems."**

Base design. In general, the devices in Figure 1, 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 performance. 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 [14]. 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.

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 2: 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).

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 2 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 3, 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 operation[2][3] (step 2) and then leveraging it to periodically inform the control panel about the produced energy (step 3). The same operation will make this information available over the Internet, *e.g.*, to the energy provider (step 4 and 5).[4]

The next sections describe how Erlang provides a stepping stone towards developing IoT applications exemplified by the smart-home scenario, and how developers use ELIoT to implement the design above.

3. Erlang

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

The example code in Figure 4 shows a recursive function that waits for incoming messages, processes them, and returns the result to the original sender. Erlang's concurrency model follows the actor model [15]: Erlang processes are named entities that do not share data, but communicate only through asynchronous message

[2] **GPaolo: Preferirei forse "functionality". Cambiamo qui e in figura?**

[3] **Luca: It's also in several other places in the paper.**

[4] **Luca: It's unclear who is the user of the added operation. It looks like the panel uses REST also to talk to the control panel, but I don't see why given the solar panel may be an ELIoT device also. Anyways, we don't say what kind of device is on the solar panel. Am I missing something?**

```
% Simple function returning the double of the 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
        {message_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
        {message_type_2, SenderPID, Content} ->
        [...]
    end,
    % Recursive call to parse next message in queue (or wait for a new message to arrive)
    loop()
end.
```

Figure 4: Erlang code sample.

passing. The **receive** statement in line 8 of Figure 4 takes the first message from the process' incoming queue, while the **!** operator is used in line 14 to return 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.

Distinguishing between message types is specified declaratively using *pattern matching*, namely, by stating constraints on the message format, as in line 10 and 16. Erlang's pattern matching also allows parsing and filtering 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.

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 and based on bytecode sent 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 the actor model described above. Developers may thus 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. One of the key features is the notion of *supervisor* process, 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 handy for dealing with localized control loops.

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 next.

4. ELIoT: Syntax and Semantics

Here we describe ELIoT's dedicated language constructs, which concerns three key aspects of inter-process communication key in developing IoT applications: handling different communication guarantees, supporting extended addressing schemes, and providing access to low-level information from the underlying communication protocols[5].

Running example. To make our explanation concrete, we consider the smart-home application introduced above. The ELIoT code in Figure 5 reports part of the implementation of the core functionality at 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 installing of the executable model of an appliance's expected energy consumption, as per scenario **B** (lines 46 to 54).

Specifically, after defining constants and structured types, the code in Figure 5 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.

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

[5] **Luca: The only example we have talks about info from the underlying hardware, not protocols.**

```
% 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 5: Excerpt of control panel code.

messages among Erlang VMs is reliable[6]. This is a strong assumption in the IoT scenarios we target, where wireless communication, often involving direct links[7] among nodes, is the rule more than the exception. At the same time, several IoT

[6] Mainstream Erlang implementations use TCP to provide such guarantees.

[7] **Luca: What do we mean by (wireless) direct link?**

```
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 6: Failure handling triggered by a failed message send.

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 5: after creating the single byte beacon (line 22), the control panel sends it unreliably using the **∼** operator.

Besides adding the **∼** operator, ELIoT addresses possible faults of the underlying communication protocol by slightly changing the behavior of the **!** operator. In presence of communication faults that cannot be resolved, ELIoT 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 6. 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 providing programming flexibility with the reality of unreliable wireless communications[8][9].

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

[8] **GPaolo: Io taglierei quest'ultima frase.**

[9] **Luca: Why?**

messages.

Addressing schemes. 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. This makes it hard 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.

ELIoT 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[10]. We use this feature to implement discovery of new appliances in Figure 5 (line 25).

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 within 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 true. We show an example of scoping filters, together with the ELIoT-specific **spawn_cond** primitive, in Figure 7.

Access low-level information from the underlying protocols.[11] Full isolation of the various layers that build a networking stack is sometimes impossible to achieve and often not beneficial to the application. Indeed, some form of cross-layering is often beneficial for overall efficiency, especially in presence of embedded devices

[10] The notion of reachability is a function of the target network scenario. Current ELIoT prototype implements the sending to **all** by using broadcast UDP; thus, the span of message spreading depends on the underlying network configuration.

[11] **Luca: This only talks about RSSI.**

```
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 7: Scoping filters.

and wireless communication, which are the norm for IoT scenarios.

ELIoT makes these considerations concrete by exposing information coming from the underlying communication layers into the receiver's incoming queue. In particular, current ELIoT prototype exposes the address of the source node and the Received Signal Strength Indicator (RSSI) coming from the radio, but the same mechanism can be used for other information. Line 31 of Figure 5 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 source address and RSSI reading in ELIoT are treated as any other type of data, and automatically materialized by ELIoT 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 [16] required for location-aware services.

5. ELIoT: System Support

ELIoT provides three dedicated system functionality to effectively support development of IoT applications: a reconfigurable REST interface for ELIoT nodes, a lightweight VM that implements the language constructs we add to Erlang, and a dedicated simulator for testing and debugging.

5.1. REST Interface

Scenario **C** in the smart-home application is supported by the *reconfigurable* REST interface exported by ELIoT nodes. It provides two features that are useful for IoT applications.

First, it allows ELIoT programmers to easily implement a REST interface for accessing their devices. This enables interactions based on standard protocols and inter-operable message formats. For example, any web browser can be used to query sensors attached to an ELIoT node, with no ad-hoc programming required.

The second feature allows programmers to extend a pre-defined REST interface by dynamically installing new REST operations on a running ELIoT node. For example, upon installation of the solar panel of scenario **C**, the attached ELIoT device will deploy an additional function onto the control panel to enrich its REST interface with a new operation: one that allows interested parties to access information about the amount of energy generated by the home. This operation will be periodically invoked by the energy provider, that will be able to access such data in a platform-independent manner, facilitating interoperability between different technologies and vendors. Notice that this kind of dynamic reconfiguration is enabled by the ELIoT's ability of spawning new processes based on binary code received from the network.

5.2. Virtual Machine

Erlang was originally designed to run on embedded platforms. However, over time 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. Most of these features find limited application in IoT applications, unnecessarily increasing the hardware requirements of the Erlang VM.

To address this issue, we develop a custom VM for ELIoT, wiping off most functionality not required in our target applications. For example, we remove several libraries, like those to support Corba middleware systems.

At the communication layer, the ELIoT VM includes 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 abandon TCP in favor of UDP[12], both for the reliable and unreliable communication and coordination primitives, namely, for sending messages in unicast and broadcast, but also for spawning processes. We implement our own reliability layer on top of UDP, supporting the **`nack`** mechanisms described in Section 4, and we also fully drop Erlang's network overlay: a logical topology that Erlang nodes automatically build as they discover each other, which requires periodic network traffic for maintenance and significant amounts of network state in memory. In ELIoT, similar functionality are delegated to the application layer,

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

only if and when required.

As a result of this work, our custom ELIoT VM drastically reduces the hardware requirements compared to Erlang's VMs, especially w.r.t. memory consumption. This enables ELIoT to run on devices that are quite unusual from those that Erlang typically applies to. We test two such platforms: *i)* a Raspberry Pi board equipped with 256 MB of RAM and an external SD card, and *ii)* 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.

5.3. 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 a key issue [17]. ELIoT offers a great opportunity to overcome this situation. By leveraging Erlang's blurred distinction between local and distributed functionality, we develop a custom simulator that allows:

- to simulate an entire system by instantiating a set of virtual nodes running *unmodified* ELIoT code;
- to model communication between nodes according to *real* wireless traces for increased fidelity[13];
- to *interact* with the simulation, if required, via a standard Erlang shell, e.g., to proactively inject messages;
- to run a *mixed deployment* where virtual nodes seamlessly interact with physical devices[14].

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. 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 will deploy.

[13] We use the traces from the TOSSIM simulator [18], although using different traces would only require developing the needed model translation.

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

6. Evaluation

ELIoT aims at simplifying IoT software development. As such, we must evaluate ELIoT by considering two aspects: the benefits it brings to developers' productivity, and the run-time overhead it introduces to provide such benefits. To quantitatively analyze these two aspects on a concrete case, we compare the ELIoT implementation of the smart-home application against a C implementation that realizes the same core functionality using the *pthread* library for multi-processing and standard UDP sockets for communication. The latter largely reflects the current practice in programming networked embedded systems [19].

6.1. Benefits to IoT Software Development

ELIoT provides two benefits to programmers: it increases their productivity by rising the level of abstraction w.r.t. 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.

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 w.r.t. 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 5. 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 46 in Figure 5 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

```c
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 }
```

(a) C implementation.

```erlang
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]);
```

(b) ELIoT implementation.

Figure 8: Deserializing appliance operating parameters in the smart-home application.

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[15]. 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.

Figure 8 provides additional insights into the expressive power of ELIoT. In

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

particular, it focuses on deserializing the operating parameters of a newly discovered appliance (see line 36 of Figure 5). In C, as shown in Figure 8a, 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 8b. In particular, the **decode_params** function in line 3 of Figure 8b 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 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 list of decoded parameters during the recursive call in line 17. Overall, the 25 lines of C code in Figure 8a reduce to 7 lines of (uncommented) ELIoT code in Figure 8b.

Similar benefits are found in creating and sending messages. For instance, Figure 9 shows the C code necessary to prepare and broadcast a beacon message, as done in lines 22 and 25 of Figure 5. 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 [20, 21].

Because of its Erlang core, ELIoT also simplifies implementing concur-

```
 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 9: C code for sending beacon messages in the smart-home application—functionally equivalent to lines 22 and 25 in Figure 5.

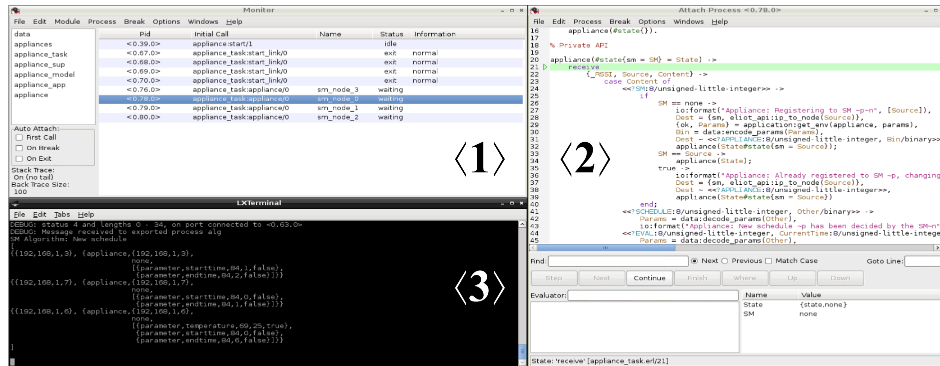

Figure 10: Simulator UI.

rent functionality, by virtue of dedicated language and system support to multi-threading. 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 access 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.

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 mixed configurations of real and simulated nodes.

Figure 10 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 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 10, 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 schedule computation. 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.

The ELIoT simulator presents functionality to developers that are rarely available using mainstream programming platforms for networked embedded systems.

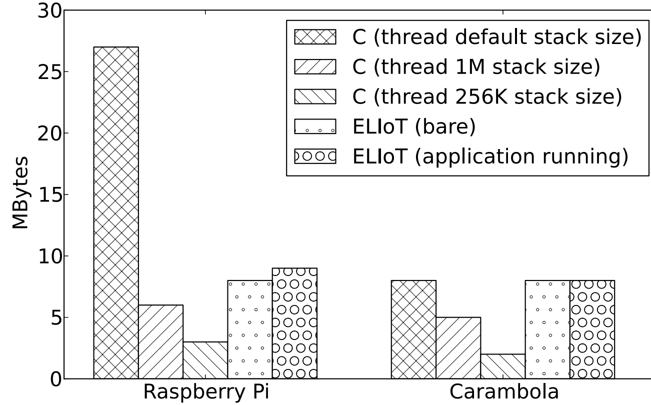

Figure 11: Memory consumption (pmap).

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.

6.2. 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.

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 11 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. Overprovisioning 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[16] would build an application that uses the same or more memory

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

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 11, 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 12 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

MB for Carambola (OpenWrt).

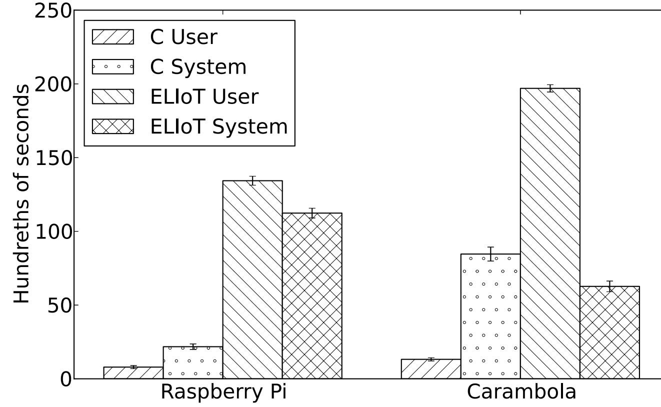

Figure 12: CPU times.

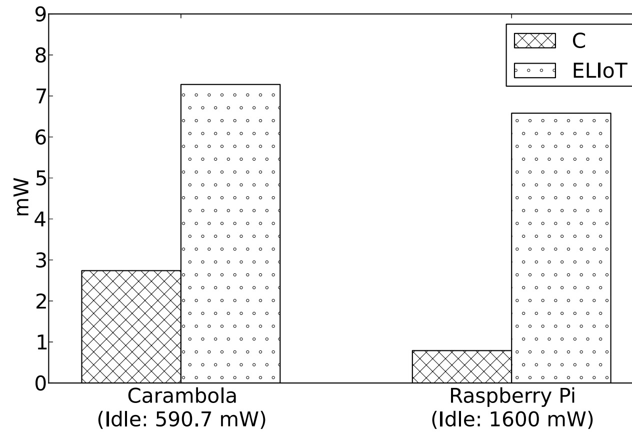

Figure 13: Power consumption. (The idle power consumption is factored out.)

generator/multimeter to measure their average power consumption throughout a single application iteration.

Figure 13 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 [22], and 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 (our own measurements).[17]

Network traffic. Using a standard network inspection tool, we measure 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.

The total overhead of ELIoT w.r.t. 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.

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. 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 time goes from 50 ms on the Raspberry Pi to less than 20 ms on the Carambola: arguably acceptable in most practical scenarios.

7. Related Work

Works closely related to ELIoT mainly target IoT software architectures and IoT application frameworks. We illustrate these next by contrasting them with our work. From a conceptual standpoint, the body of work on sensor network programming and pervasive computing in general is also worth mentioning due to similarities in some of the objectives. We complement the discussion with a brief analysis of application-specific frameworks.

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 [23] exploits an IPv6-based network layer to build CoAP services atop. The IoT-A project [24] defines an architectural reference model for the interoperability of IoT devices, whereas Spitfire [25] investigates unified concepts for facilitating the effective development of

[17] **Luca: I'd cut entirely the Kindle stuff. It doesn't add much IMO.**

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 [26], shared memory systems [27], and platforms providing storage and processing facilities for sensor data, such as Cosm [10]. Different solutions instead provide remote access to sensors and actuators from the Internet, such as sMAP [28] and CoAP [29].

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

There also exist works tackling the development process of IoT applications. Srijan [30], 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.

Sensor network programming and pervasive computing. Although based on a different target hardware, existing solutions for sensor network programming [14] allow an efficient implementation of localized interactions by deploying the application logic right onto the embedded devices. Sensor network programming may occur at the operating system level [31, 32], based on platform specific APIs [33], by relying on custom virtual machines [34], or by using higher-level abstractions [14]. Largely common to these approaches, however, is 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. From a conceptual standpoint, ELIoT aims at bringing the localized interactions already enabled by sensor network programming in Internet-connected embedded networks.

Similar considerations apply in relating ELIoT to the body of work in pervasive computing systems. For example, Aura [35] and Gaia [36] focus on effective development of interactions between users and the devices they operate; MundoCore [37] 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.

Application scenarios. We use a smart-home application to exemplify the use of ELIoT. Ad-hoc solutions exist for developing software in specific application domains. For example, Gator Tech [38] presents the design of a pervasive computing system especially conceived for elderly people, within an environment enriched by sensors and actuators; whereas HomeOS [39] 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 [7]. 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.

8. Conclusions

We presented ELIoT, a development platform for the IoT that allows to combine localized and Internet-scale 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. At system-level, ELIoT nodes export reconfigurable REST interfaces for standard-compliant interactions, while a dedicated VM tailored to the typical 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.

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

References

[1] L. Atzori, A. Iera, G. Morabito, The Internet of Things: A survey, Computer Networks 54 (15).

[2] 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.

[3] 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).

[4] D. Uckelmann, M. Harrison, F. Michahelles, Architecting the Internet of Things, Springer, 2011.

[5] 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.

[6] 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.

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

[8] BeagleBoard, `beagleboard.org/Products/BeagleBone`.

[9] Raspberry PI, `www.raspberrypi.org`.

[10] Cosm, `cosm.com`.

[11] 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.

[12] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007.

[13] A. Sivieri, L. Mottola, G. Cugola, Drop the phone and talk to the physical world: Programming the internet of things with Erlang, in: SESENA, 2012, pp. 8–14.

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

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

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

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

[18] 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.

[19] M. Barr, A. Massa, Programming Embedded Systems, O'Relly Media, 2006.

[20] U. Wiger, G. Ask, K. Boortz, World-class product certification using erlang, SIGPLAN Not. 37 (12).
[21] B. J. MacLennan, Functional programming: practice and theory, Addison-Wesley Longman Publishing Co., Inc., 1990.
[22] A. Carroll, G. Heiser, An analysis of power consumption in a smartphone, in: Proc. of the USENIX annual technical conference, 2010.
[23] IoT6 - Universal Integration of the IoT, www.iot6.eu.
[24] Internet of Things - Architecture, www.iot-a.eu.
[25] Spitfire Semantic Web interaction with Real Objects, spitfire-project.eu.
[26] 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.
[27] 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.
[28] 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.
[29] Z. Shelby, K. Hartke, C. Bormann, B. Frank, Constrained application protocol (CoAP), draft-ietf-corecoap-07 (2011).
[30] 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.
[31] 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.
[32] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, System architecture directions for networked sensors, SIGPLAN Not. 35 (11) (2000) 93–104.
[33] Waspmote, www.libelium.com/waspmote.
[34] 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.
[35] 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 - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, WICSA 3, Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 2002, pp. 29–43.
[36] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, K. Nahrstedt, A middleware infrastructure for active spaces, IEEE Pervasive Computing (2002) 74–83.
[37] E. Aitenbichler, J. Kangasharju, M. Mühlhäuser, MundoCore: A light-weight infrastructure for pervasive computing, Pervasive and Mobile Computing (2007) 332–361.
[38] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, E. Jansen, The gator tech smart house: A programmable pervasive space, Computer (2005) 50–60.
[39] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, P. Bahl, An operat-

ing system for the home, in: Proc. 9th USENIX Conf. on Networked Systems Design and Implementation, 2012.