D-LITe : Distributed Logic for Internet of Things sErvices

Sylvain Cherrier*, Yacine M. Ghamri-Doudane†,Stéphane Lohier* and Gilles Roussel*
* *Université Paris-Est , Institut Gaspard Monge (LIGM)*
77454 Marne-la-Vallée Cedex 2
Email: [firstname.lastname]@univ-paris-est.fr
† *Ecole Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (ENSIIE)*
1 square de la résistance,91025 Evry Cedex
Email: yacine.ghamri@ensiie.fr

***Abstract*—Smartphones, PDA, Sensors, Actuators, Phidgets and Smart Objects (i.e. objects with processing and networking capabilities) are more and more present in everyday's life. Merging all these technologies with the Internet is often described as 'Internet of Things' (IoT). In the IoT vision, Things around us provide a pervasive network of interacting and interconnected devices. However building IoT applications is a long and arduous work, reserved for specialists, requiring specific knowledges in terms of network protocols and programming languages. The lack of widespread and easy-to-configure solutions is an obstacle for the development of this area. A universal framework, offering simplification and standardization, could facilitate the emergence of this promising field in terms of applications and business. IoT needs a solid foundation for rapid, simple development and deployment of new services. In this paper, we present D-LITe, a universal framework for building IoT applications over heterogeneous sets of small devices. D-LITe offers solutions for deploying application's logic, and executing it on Smart Objects despite their heterogeneity. An implementation of D-LITe on tiny devices, such as TelosB motes, allows to show that our framework is realistic even with the constraints of such devices.**

***Keywords*-Web of Things; Services Choreography Architecture; Distributed logic;**

I. Introduction

Objects become more and more clever and interacting devices. Manufacturers introduce processing power and networking technologies in common objects leading to the concept of *Smart Objects* and to *IoT* or *Web of Things* (WoT is the web version of IoT, easier to use for end-users). In this paradigm, *Things* offer a digital environment, sensing and acting on real world. Users are able to deal with their digital environment. "Home automation" is an example of IoT, in which people organize services offered by things present in their living environment. But there is a main issue that still prevents the raise and wide deployment of IoT: The multitude of Smart Objects (such as Sensors) uses different languages (C, NesC, Java...), different Application Programming Interfaces (Arduino, ZigBee Application...), different Operating Systems (TinyOS, Contiki...) through different network protocol stacks (IEEE 802.15.4, Zigbee, 6LowPAN) that may be mutually incompatible unlike the widely spread IP Network used by more powerful objects. Creating an application dealing with each kind of smart objects becomes a specific work, performed for a specific type of hardware (Operating System, Network technology) and with specific programming tools (languages, API). It involves the need of a gateway to be accessed from the Internet and to communicate with other objects. Another issue is the deployment of applications, mainly consisting of ROM flashing on each smart object, that requires human intervention and manipulation. This leads to important time and cost overhead.

Creating IoT applications is complex and time-consuming, hardware dependant, and hardly scalable. In this paper we intend to solve these problems by proposing a universal framework and architecture: D-LITe, a new Distributed Logic for Internet of Things sErvices creation and deployment. D-LITe allows to design simple, scalable and easy-to-maintain applications and deploys them over heterogeneous platforms.

The reminder of this paper is organized as follows: First, we present the related works (Section II) and the background (Section III) attached to our solution. The overall design of D-LITe is described in Section IV, while Section V focuses on the protocols and languages used by D-LITe. Section VI deals with the Implementation and Validation of D-LITe. Finally, concluding remarks and future research directions are given.

II. Related Works

Internet of Things has many definitions [6]. The IoT paradigm incorporates other technologies such as pervasive or ubiquitous computing as well as ambient intelligence (AmI) [7], [12]. To realize IoT applications, programmers or users have to deal with multiple devices that are not interoperable. There are many approaches on how to program such network applications. We consider macro programming as described in [17] "*programming the sensor network as a whole, rather than writing low-level software to drive individual nodes*". Many objects that think come with processing capabilities, but no code to use it. "*for years, closed networks*" were "*deployed for a specific application... we argue that the next generation WSN require customizable*

architecture" [24]. Giving every node the ability to interact with any other seems to be a solid basis for building distributed applications. Authors in [24] propose to give standard access to nodes to offer such a customizable architecture.

Every node in IoT applications should be reachable and usable. However no common architecture is provided. ZigBee Alliance [4] has developed adequate protocols for Sensors. ZigBee is a complete solution, based on the use of the IEEE 802.15.4 at the lower layer. It defines the reminder of the network architecture up to services (called *ZigBee Profiles*). For example, ZigBee Home Automation is one of those Profiles "*enabling smart homes that can control appliances, lighting, environment, energy management, and security as well as expand to connect with other ZigBee network*" [5]. Nevertheless many technologies (SmartPhone, PC, sensors, actuators...) are involved in Home Automation so that a gateway is mandatory to connect to other networks (Internet). An end-to-end communication could be degraded by such a gateway. Protocol's conduct, exchanges between nodes, the size of exchanged messages can be so different that their translation may be particularly difficult. The specialized protocol's dynamics on one side may be unsuitable for the other side. All these differences can be difficult to solve, or just very penalizing in terms of adaptation, effectiveness, and response time.

Dynamicity, scalability and reconfiguration are also issues. Users may want to use the computing capabilities of smart objects and take advantage of the versatility of programmable devices. Changing the interactions of household objects when integrating a new device, or simply changing the behaviour of the total application is an expected asset of the IoT. Until now human interventions are still required to set and update nodes. Reprogramming "over the air" (OAP, Over the Air Protocol) answers that issue [30]. OAP is proposed in SYNAPSE [25], Deluge [18] or Dynamic TinyOS [22]. SYNAPSE and DELUGE mainly focus on how to organize a reliable transfer on a non-reliable wireless network, while Dynamic TinyOS deals with efficient software updating, but is strongly coupled with TinyOS Operating System.

Our aim is to provide a solution loosely coupled to network protocol stack, operating system, language and hardware.

III. Background and Vision

Such as Generic Virtual Machine for a high-level programming language[1], D-LITe constitutes a basic framework for building simple and universal applications. Many concepts of quite distant areas are melt in D-LITe to give the end-user a simple way to design and deploy logical applications on nodes.

[1]JVM for Java, for example, or Parrot for Perl

A. From Internet to Smart Objects

D-LITe nodes need to be accessed from the Internet. IPv6 Protocol seems to be a good candidate for that purpose, because this standard is most likely to be used to deal with billions of nodes. By using header compression mechanisms, 6LowPAN [27] proposes a solution for IPv6 compatibility over IEEE 802.15.4 networks. It gives universal access to data collected by sensors and actions done by actuators. Even if 6LowPAN is restricted to support only UDP, nodes are able to offer all kind of Web's well-known services because it uses IP. 6LowPAN turns motes from connected data collectors into real small data servers.

B. Accessing services : SOAP, REST and CoAP

In an end-to-end communication, motes can be considered as service providers. To access the provided services, Service Oriented Architecture (SOA) is a well-known solution [14]. The idea of using such paradigm for Sensor Network is presented in TinySOA [24]. SOA introduces loose coupling between services and applications as well as hardware independence. Many protocols realize SOA. One of them is SOAP [3], but it is a very verbose protocol (i.e. consuming bandwidth and requiring important processing). Sensors Networks have a very limited bandwidth, that is why D-LITe is organized according to REST approach. REST architecture [15] is an alternative to SOAP for distributed applications, and has many advantages. Using standard HTTP methods, REST is lightweight and simple to adapt to our purpose. However a major issue remains: because of smart objects' memory size, TCP and moreover HTTP (needed by REST architecture) are very hard to fit [6] in constrained devices. To address this issue, CoAP [26] offers the same characteristics as REST: CoAP *"extends the REST architecture to a suitable form for the most constrained nodes"* of Sensors Network [28]. Furthermore, CoAP is build over 6LowPAN, and already exists in Contiki operating system [11] for Wireless Sensor Networks. By implementing HTTP over UDP, and using compression of HTTP methods, CoAP is designed to simply permit translations between standard and universal REST commands from the Internet and a 6LowPAN Network, while being particularly suitable to the limited payload of smart objects.

C. Services : Choreography and Orchestration

We consider that an important part of IoT applications can be designed as a collaboration between nodes. The whole application's logic can be spread into small autonomous part on each node. To combine services offered by motes (data collected or possible actions), SOA is divided in two approaches : Services Orchestration or Services Choreography [23], [10]. They mainly differ in the centralized approach of orchestration compared to the collaborative form of choreography. D-LITe uses the Choreography concept. In our Choreography, there is no central controller, each node

is autonomous. The node knows what to do, and reacts to context's change. In D-LITe's choreography, each node is like a dancer. Each dancer knows his steps, and reacts on events of the very near environment. There is no centralized control of any supervisor ; decisions are mainly made at the lowest level. On the contrary, in the usual definition of services orchestration, a central point would control all exchanges. The central point would call the services offered by nodes and compute results. No nodes would act on its own. Because it uses choreography, D-LITe delegates small parts of the global application to each participant, using processing capacities closer to the needs, saving bandwith and therefore energy.

D. Designs Patterns used in D-LITe

D-LITe uses Gang of Four (GoF) [16] Design Pattern (DP) Observer and Strategy. Some protocols propose Observer DP in Sensor Networks. Using such protocols, nodes can subscribe to others that publish data as in mqtt [20] or in TinyCops [19]. However mqtt is not based on 6LowPAN but on Zigbee, and TinyCops is a TinyOS module. Consequently they are not usable in a heterogeneous environment. D-LITe implements this DP on its own.
Strategy DP dynamically changes an object's behaviour. Basically, Strategy delegates one object's logic to another object, chosen inside a set of objects each implementing a different version of the same command. This can be managed and changed "on the fly".
D-LITe is inspired by Strategy. D-LITe installs a static piece of code on each node. That code offers an access to a dynamic part of the application that can be configured or changed. This dynamic part is under the control of a rule analyzer. The rule analyzer can execute a logical description of node's expected behaviour depicted by a "set of rules". This logical description is variable, can be set through the network and dynamically changed.

E. Finite State Transducers (FST)

To describe our choreography, a tool to program each node is required. As presented in Section IV-C, we chose to use macro-programming approach shown for example in [29], [21]. D-LITe uses Finite State Transducers (FSTs) to describe the application's logic. Describing this logic with an Automaton rather than a programming language is somewhat limited, but has absolute advantages: universality, very low memory footprint for the parser, and very concise expression of the description. Automata are hardware independent, text-based, and easy to learn. FST are Finite State Machines (FSM) with an additional output Alphabet. They are often used in Natural Language Processing. In D-LITe, input and output alphabets are the messages exchanged by nodes through the network. States are the node's reaction to received messages. The idea of using a Transducer to program a sensor, and the rule's approach, were inspired by J. Baliosian and al papers [8], [9].

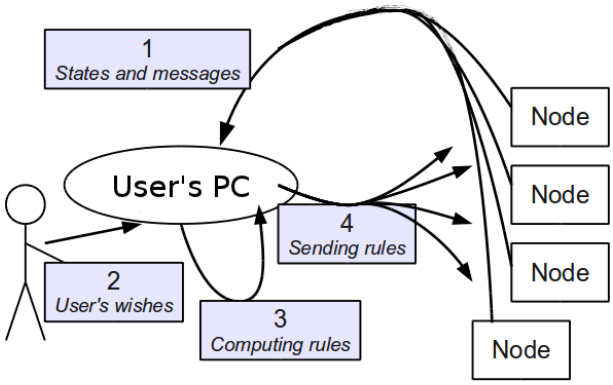

Figure 1. D-LITE overall architecture : User can discover all nodes capabilities then describe his needs by creating rules. D-LITe deploys these rules that each node follows.

IV. D-LITe : An Architecture to Deploy Application's Logic

D-LITE is organized to enable the writing of small cooperating units realizing an application, like the cells of a spreadsheet are used in end-user development.

A. Overview of D-LITe Distributed Framework

D-LITe is a distributed framework for realizing IoT applications. It consists in building applications as a collaboration of smaller logical units. A mote[2] is more than a simple sensor or actuator. Because of its small computing capabilities, this kind of node can do additional processing. For this purpose, D-LITe is installed on each node. As it uses standard protocols (IPv6 and REST), D-LITe offers a universal access, hiding specificity of the different hardwares used. The REST access given by D-LITe is used to deploy orders (configuration, FST) on each node. As shown in Figure 1:

1) An end-user collects information about nodes capabilities.
2) He expresses his need : he describes a sequence of interactions between elements.
3) This sequence is then transformed as a set of FSTs (one FST per node).
4) Each node will receive its own FST and other configuration information.

The D-LITe architecture allows an end-user to transmit rules and configuration to each node. Each D-LITe enabled node contains a rules analyzer to execute the FST. D-LITe nodes also have a messaging service to interact with each other (Figure 2).

[2]Like Crossbow TelosB or Imote, Oracle SunSpot, or Aduino Uno. D-LITe is mainly design for motes, even if some more powerful hardwares are supported

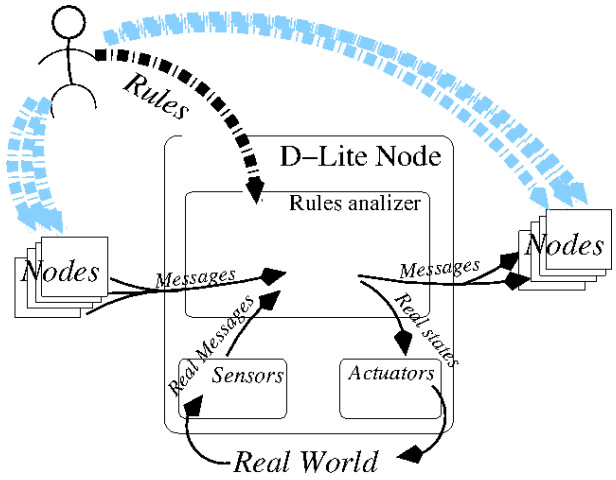

Figure 2. Each D-LITe node received its rules ; from now it is ready to execute them. A node obeys the FST's rules according to messages it receives from other nodes or from its own hardware (sensors). It changes state (that may impact the real world if it's an actuator) and sends messages to its observers. These will react to that message.

B. Distributed Application Choreography

D-LITe is based on the idea that Internet of Things applications can often be seen as a Finite State Machines choreography. D-LITe enables to depict each part of the application's logic as FSTs which will be executed in several nodes. Each FST (a set of rules) can be dynamically and quickly send to the proper node. An end-user has to organize his thoughts to describe his application as a choreography of Transducers, just like he organizes his formulas in each cell of a spreadsheet (Figure 1). When a node receives a message or changes state, it affects other nodes, just like cells in a spreadsheet react to changes in other cells; Updating their content results in a chain reaction on depending cells (Figure 2).

As choreography starts, every node may receive a message, because something has happened. That message is inspected by the algorithm in charge of the FST's execution. If a rule matches the current state and this received message, the node's state changes, and the output message defined in the rule is sent to Observers.

C. Node's Logic Representation using Transducers

A Finite State Transducer has a formal representation as a 6-tuple $T(Q, \Sigma, \Gamma, I, F, \delta)$. D-LITe defines the meaning of each element as follow:

- Q represents all *States* for a particular node,
- Σ are *Input Messages* handled by a particular node,
- Γ are *Output Messages* a particular node can send,
- I is the *Initial State* (only one in D-LITe),
- F stands for *Final States*,
- δ contains transitions (which are our "*set of rules*").

ϵ element stands for empty. The main adaptation introduced in D-LITe is *Input Messages* and *Output Messages* in place of alphabets. Figures 3, 4 and 5 are simple examples of applications, using two types of nodes : switches and lights.

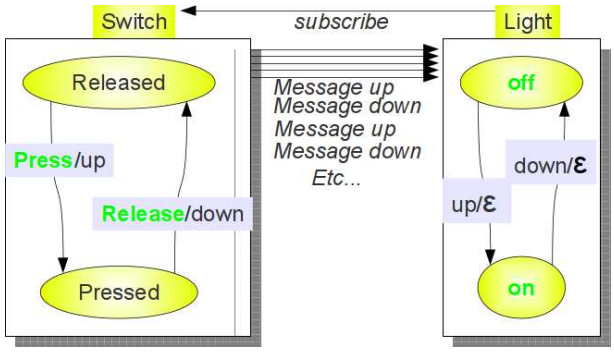

Figure 3. Example 1: a simple node for lighting. Press and Release are real Messages. On and Off are real States.

1) Example 1: a switch and a light: In the very simple example given in Figure 3, a switch (sensor) and a light (actuator) can be set this way: When the light receives a "up" message, it moves to "ON" state. When it receives a "down" message, it sets its state to "OFF". When somebody presses the switch button, its state moves to "Pressed", and it sends an "up" message. Similarly when somebody presses it again, it moves to "Released" state , and it sends a "down" message. The two FST representing this two nodes are:

- for the switch: $Q = (\text{“}Pressed\text{”}, \text{“}Released\text{”})$, $\Sigma = (\text{“}Press\text{”}, \text{“}Release\text{”})$, $\Gamma = (\text{“}up\text{”}, \text{“}down\text{”})$, $I = (\text{“}Released\text{”})$, $F = (\epsilon)$ and δ is described in Figure 3.
- For the light: $Q = (\text{“}on\text{”}, \text{“}off\text{”})$, $\Sigma = (\text{“}up\text{”}, \text{“}down\text{”})$, $\Gamma = (\epsilon)$, $I = (\text{“}off\text{”})$, $F = (\epsilon)$ and δ is described in Figure 3.

This is the way an end-user can simply program a standard switch/light pair.

2) Example 2: Introducing a new state: Figure 4 introduces a new behaviour not planned in the process of light switching : a delay. Our application offers a Time service (a time message is send every 10 seconds). When receiving the "down" message, the light stays on, and waits for a "time" message from the Time service. On receiving this "time" message, the light switches off. To realize this feature, we introduce a logical state on the on/off light process : a Wait State.

This state has no physical action but represents the fact that the light is now waiting for another message. This is a *logical state*. After receiving this time event, the light moves to the "Off" state, and really switches off. To implement this example, the switch FST remains unchanged. However, the light FST becomes : $Q = (\text{“}on\text{”}, \text{“}off\text{”}, \text{“}WaitState\text{”})$, $\Sigma = (\text{“}up\text{”}, \text{“}down\text{”}, \text{“}time\text{”})$, $\Gamma = (\epsilon)$, $I = (\text{“}off\text{”})$, $F = (\epsilon)$ and δ is described in Figure 4. We also add the light as an observer of Time service.

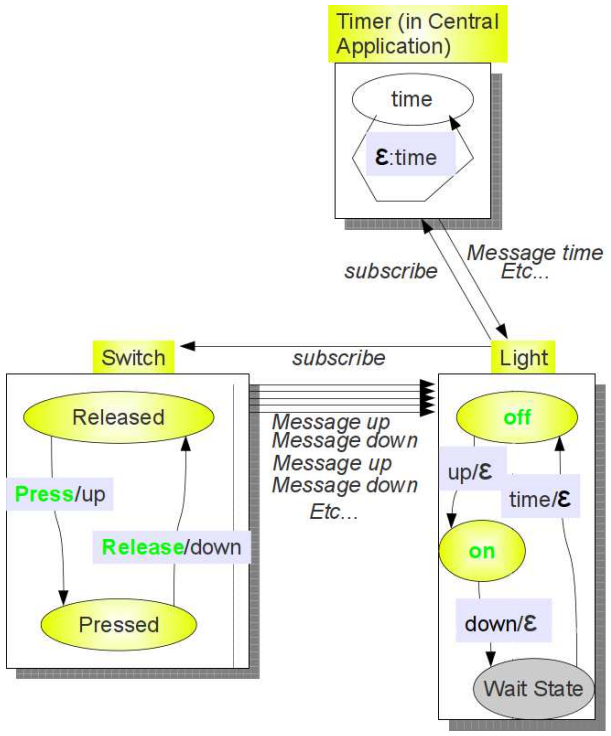

Figure 4. Example 2, introducing a logical state : Wait State (On and Off are real states of light actuator).

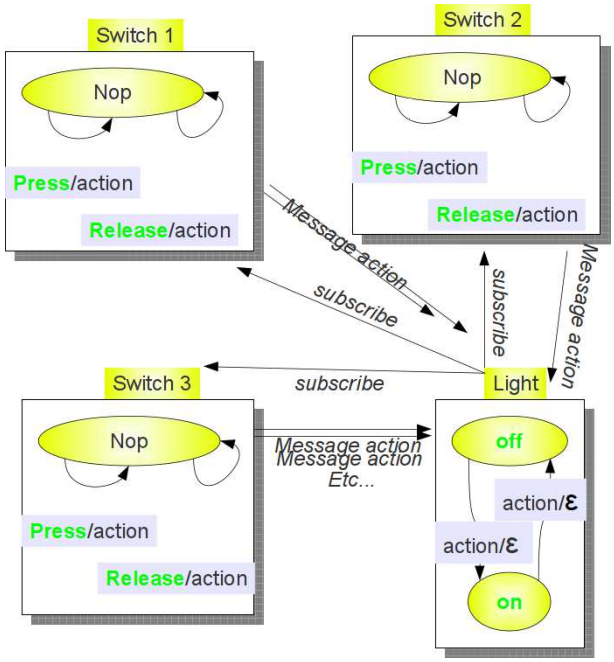

Figure 5. Example 3, losing semantic on switches to solve a 3-way switching problem.

3) Example 3: a semantic loss: Figure 5 represents a 3-way (or more) switching. The user merely needs to define a single message (for example "action") to be sent by each switch when the button is used (vs. two messages in previous examples). No matter how the button is now (pressed or released), the light's state has to change.
For this purpose, the user explains that "Press" or "Release" events on the switch send a unique message : "action" (a poor semantic message). There is no other state needed on the switch. The light subscribes to all switches, and each switch sends only "action" message when pressed or released. By receiving the "action" message, the light changes state from "on" to "off" and vice versa. The corresponding two FSTs are:

- For each switch: $Q = (\text{“}Nop\text{”})$, $\Sigma = (\text{“}Press\text{”}, \text{“}Release\text{”})$, $\Gamma = (\text{“}action\text{”})$, $I = (\text{“}Nop\text{”})$, $F = (\epsilon)$ and δ is described in Figure 5.
- For the light: $Q = (\text{“}on\text{”}, \text{“}off\text{”})$, $\Sigma = (\text{“}action\text{”})$, $\Gamma = (\epsilon)$, $I = (\text{“}off\text{”})$, $F = (\epsilon)$ and δ is described in Figure 5.

D. Specific States and Messages

6LowPAN and CoAP make our node able to communicate with others, and to be dynamically configured. The use of FST is a simple way to express a sequence of logical actions. But in spite of these capabilities, our architecture is not really sensing and acting on the real world. This is only logic. To make a link between D-LITe and the real environment, we propose two Types of Messages and States : *Real* or *Logical* (Figure 2)
Logical messages or states are useful for reasoning. For example, if we want a light not to switch off immediately when we push a button, we can introduce a logical state : "waiting" (see Figure 4). This state has no impact on the real world but merely means that someone starts the process of switching off. Then a *Real State* is used. "on" and "off" are such states. When the FST moves to them, the light really switches "on" or "off". In Figure 4, rules explain that on receiving the time message while being in "waiting" state, the light really switches off by going in "off" state.
Messages are treated the same way. Many of them are *logical messages* defined by the user to describe his logical steps in the sequence of actions. The others are *real messages* sent by the hardware (i.e. the sensing part of the sensor). For example, "Press" and "Release" (cf. Figure 3, 4 and 5) are *Real Message* sent by hardware to its FST each time someone uses the switch's button.
The only interaction between D-LITe and hardware comes from the notion of *Real Messages and Real States*. That is why D-LITe is loosely coupled to hardware. Thus, in our FST $T(Q, \Sigma, \Gamma, I, F, \delta)$, few elements of Q and Σ are in contact with the real world. All *Real Messages and Real States* are detected during the discovery phase. In Figure 3, 4 and 5, *Real States and Real Messages* are written in green and bold.

Table I
SALT ORDERS : SALT HAS 2 MAIN PURPOSES. THE FIRST (ORDER) DESCRIBES NODE'S LOGIC. THE SECOND (INPUT) ANNOUNCES A MESSAGE.

variable	value	with	description
order	*init*	state=xxx	must initialize state to 'xxx'
order	*rule*	Rule Message	A rule the transducer must obey: see details below
order	*link*	uri=[aa:bb::cccc]	uri contains the IPv6 address of one observer
input	*xxx*		Message service : a node (or the hardware) sends 'xxx' to this node

V. D-LITE LANGUAGE (SALT)

D-LITe is organized to allow the design and the deployment of applications depicted as a choreography of logical Finite State Transducers. D-LITe proposes a description language (*SALT : Simple Application Logic description using Transducers*) to configure nodes and allow them to communicate.

A. SALT description

On each D-LITe node, rule analyzer and communication features are installed. To describe his application's logic, a user needs a language to:

- *Delete all settings*, i.e. start a new application.
- *Set the Initial State*, i.e. set FST's starting state.
- *Express each Rule*, i.e. describe the node's FST.
- *Attach Observers*, i.e. allow a node to send messages to a specific list of other nodes.

There is also other needs. A node must be able to:

- *Describe itself*, i.e. give its real messages/states during discovery phase.
- *Communicate with others*, i.e. send messages to its observers, and receive messages from other nodes.

B. SALT Messages format

SALT uses a very simple textual form to express and fulfill all the above mentioned tasks. The use of this format instead of other standardised ones, such as JSON for instance, is motivated by the fact that the parser for standardised ones are usually heavy and could not fit the node's memory limitation. Hence, we use a *name=value* form to limit bandwidth and memory consumption. Names and values should not be more than 6 characters long (on our Contiki implementation).

The format used by SALT messages is described in table I. DLITe's FST is fully described by its initial state (order is set to *init*) and the set of rules (order is set to *rule*). Observers list is given by order *link*. As the choreography starts, messages are exchanged between nodes using *input* message.

Rule's Message is a one-liner (see table II) that gives a description of each FST's transition. All states, input and output alphabets are deduced from the complete set of rules.

Table II
SALT RULES : FST (TRANSITIONS, STATES, INPUT AND OUTPUT MESSAGES) ARE DESCRIBED IN A ONE-LINER.

in *order=rule&state=xxx&msg=yyy&Nstate=zzz&Smsg=aaaa*	
variable	description
state=xxx	if current state is "xxx"...
msg=yyy	... if "yyy" message is received...
Nstate=zzz	...then the node moves to "zzz" new state...
Smsg=aaa	...and sends "aaa" message to Observers.

C. SALT Usage

SALT messages are exchanged between the end-user or a node and other nodes (Figure 2). We use CoAP [26], complying Internet standards, to have a small overhead and to be accessible from everywhere. Therefore, the Capillary Internet[3] can be reached through CoAP. Table III shows a complete list of SALT messages that are sent to a node, using CoAP's PUT method.

D. CoAP Methods

CoAP methods are used in D-LITe for following purposes:

- DELETE : Clean FST, current state, and observers list.
- GET : Obtain node's description (i.e. the Real states/messages supported by hardware).
- PUT : Give configuration's orders to the node (i.e. Initial state, observers list, and the FST's rules) (using SALT's "order" messages).
- POST : Messages service, to be managed by FST (using SALT's "input" message).

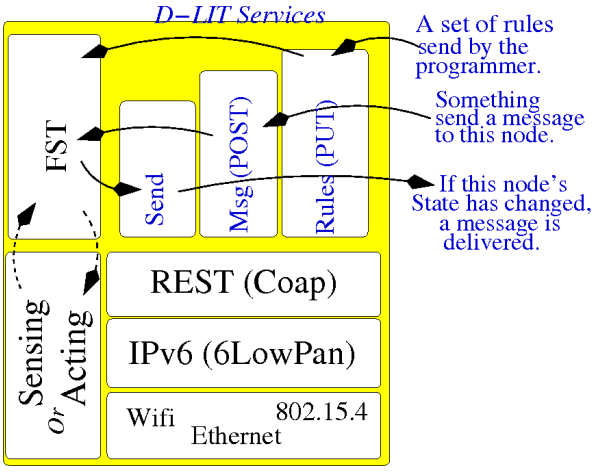

Figure 6. D-LITe Node's services. Each node has its FST sent by the user, and receives and sends messages through standard network protocols.

Table III
SALT MESSAGES FOR A D-LITE NODE : AFTER RESETTING THE NODE, THE USER SETS ITS INITIAL STATE, ITS FST, AND ITS OBSERVER

```
order=init&state=rlsd
order=rule&state=rlsd&msg=push&Nstate=prsd&Smsg=up
order=rule&state=prsd&msg=push&Nstate=rlsd&Smsg=down
order=link&uri=[fe80::f0:1:303]
```

E. An Example : Simple Configuration of a Node

Let us take the simple example of a switch controlling another device. In this case, the following SALT messages are exchanged with the switch's node: First an end-user uses the *GET* CoAP's method to retrieve information about the node, especially its real states and messages. After designing the choreography, the end-user broadcasts the logic to each node. He uses *DELETE* to clear all rules and current state on each node. Then, he sends (cf. Table III) initial state, all rules representing the FST, and Observers' list using the *PUT* method.
The switch is initialised in state "rlsd". The two rules explain that on receiving the "push" message, the switch will alternately be "rlsd" or "prsd" (in this case, "push" is a real message, generated by hardware. "rlsd" and "prsd" are logical states, defined by the user). The latest order links the switch to node fe80::f0:1:303, which is the IPv6 address of the light controlled by this switch. This light (fe80::f0:1:303) must then be configured to react to messages sent by our node (not shown in this example).
Each node is now ready, and the choreography can start. Nodes communicate with others using the *POST* method, receiving and sending messages following FST instructions, as shown in Figure 6.

VI. IMPLEMENTATION AND VALIDATION

A. Implementation on Netkit, Contiki, Cooja, and Telos B

To test our architecture, we realized a simulation using Coapy [1] to implement the services offered by a D-LITe node. Nodes were simulated on a virtual network with Netkit [4]. Our objective was to test our language.
Once virtual nodes were collaborating under Netkit with Coapy, we decided to implement our code on Contiki [13]. Contiki offers 6LowPAN, CoAP and REST implementations, and runs on real nodes like TelosB or MicaZ. It comes with Cooja, a network simulator of emulated motes. Our D-LITe implementation (Figure 7) has been tested in the Cooja emulator and on real Nodes (TelosB). Coapy is used as a client to send commands from a PC. We wrote Scripts sending initial state, rules and observers list for each example presented in Section IV-C. We also use a Firefox's plugin handling CoAP called Copper [5] to get values or send commands to nodes directly from a PC.

[3]end-to-end Internet, from everywhere to nodes
[4]a Network Simulator (http://wiki.netkit.org)
[5]https://addons.mozilla.org/en-US/firefox/addon/copper-270430/

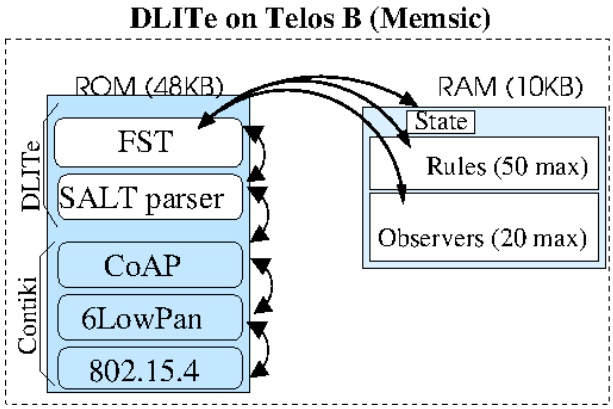

Figure 7. DLITe implementation on TelosB.

D-LITe's code uses 6LowPAN and CoAP API provided by Contiki (Figure 7). SALT messages decoding and FST rule's analyzer is implemented in our D-LITe's code. The binary size of D-LITe for a TelosB is 47KB. TelosB has 48KB program flash memory for storing programs and 10KB RAM for data. Programming TelosB is done by flashing the ROM through a USB connector. In our architecture, the software is divided in two parts. One is D-LITe framework (the fixed part) and the other is the FST's description (the variable part written by users). We flash D-LITe once on a node, and store it in ROM. No more physical contact with the node will be required. Then, each user's program (i.e. the FST's description) is sent through the network, at any time, and stored in RAM. The FST is then executed by the D-LITe framework. All manipulated data are 6 characters long. A rule's length is 24 Bytes (4 words of 6 characters, 2 states and 2 messages). Our implementation can handle up to 50 rules. Each Observer is stored in 16 Bytes (IPv6 address's size). We planned a maximum of 20 Observers. These data describing FST's behaviour represent 1526 Bytes of the 10KB RAM available [2].

B. Validation use case 1 : a simple example

This is the classical way to control an on/off device (Figure 3). We just want the switch to control light. FST's details were shown before (cf. Section IV-C).
5 orders are used to configure the switch. The first one is a call to the DELETE CoAP method. The following 4 orders are sent with the PUT method (Table III) : One to define the initial state, then the 2 rules, and finally the observer (the light).
4 orders are sent to the light device: the DELETE CoAP method to clean the FST, then the initial state (dark), and finally 2 rules (when receiving "on", go to "light" state (which is a real state), and when receiving "off", go to "dark" state (which is also a real state)).
The whole code is sent in 9 CoAP packets. SALT messages

total size is 309 bytes, which is far less than the 10KB available RAM memory. Those messages represent the whole application code. 2 rules are stored in light's RAM (48 Bytes used). 2 rules and 1 observer's address are stored in switch's RAM (64 Bytes used).

C. Validation use case 2 : Introducing a logical state

Figure 4 shows a simple example of the use of a logical state. With D-LITe, it is possible to deploy a new program that supersedes the one shown above. The switch doesn't need to be updated, so we keep it unchanged. A DELETE method is used to clean the light's FST. Initial state is set to "dark". The light needs 3 rules: One for "off" to "on", one for "on" to "Wait", and the last when receiving the "time" event to really switch "off". And finally, we send the IPv6 address of the light to the time service, as light becomes an observer of this service.
In this example, 10 CoAP packets are send on the network. Switch's RAM usage is still 64 Bytes on the switch, and 88 Bytes are used on the light.

D. Validation use case 3 : using loss of semantic

Dealing with more than two switches to control a light can be done with D-LITe. In that case, each node has just to send a signal to make the light change its state. If the light is on and someone presses the button, the light's state needs to be changed. Switch's former state and message's type do not matter. Figure 5. After deleting FSTs in each node, each switch is initiated to state "nop". 2 rules are sent to explain that "press" and "released" messages generate the same "action" output and go to "nop" state. Each node receives an order to register the light as listener. Setting the light is as simple. FST is deleted, then initial state is set to "dark", and 2 rules describe a flip-flop : "action" message changes state from "dark" to "light" and vice versa.
This application uses 4 CoAP packets to configure the light. Each switch needs 5 packets to be set. 2 rules and 1 observer's address for each switch represent 64 Bytes of node's RAM.

VII. Conclusion

D-LITe splits the application in two parts. A fixed part is installed once on each node physically (flashed on ROM for example). That part offers generic services and access. The second part is dynamically uploaded through the network. This one allows to describe the application's logic using very simple textual form (i.e. SALT). This architecture gives D-LITe some advantages. Any changes is simple and fast to deploy. No physical access to a node is needed to completely re-adapt its behaviour. The logic is not very hard to describe. It uses a textual form. It is hardware independent. Programming is based on nodes cooperation, each participating node supporting a small part of the overall application. The vision of the application is a choreography of FSTs, exchanging messages and reacting to received ones. Even if the possibilities of FSTs are restricted, our architecture covers many usual IoT's use cases. Our implementation on TelosB shows that D-LITe can run on constrained devices (48KB) (TelosB RAM's size (10KB) can store up to 50 rules and 20 observers). The D-LITe framework is easy to access and operate by standard and well-known tools as it is based on IPv6 and REST. We are already using D-LITe to test applications, and see where it can be adapted.
The main contribution of this paper was to show that it is possible to quickly and easily develop IoT applications in a standardized way, and quickly and easily spread them over any kind of hardware through the Capillary Internet. In the future, we will mainly work on how to improve the architecture, offer reliability, and give it some configuration automation.

References

[1] Coapy, a python implementation of coap. http://coapy.sourceforge.net/.

[2] D-lite website, with codes, documentations, videos, examples. http://igm.univ-mlv.fr/PASNet/.

[3] Soap version 1.2 (w3c). http://www.w3.org/2002/07/soap-translation/soap12-part0.html.

[4] Zigbee alliance. http://www.zigbee.org/.

[5] Zigbee profiles. http://www.zigbee.org/Products/CertifiedProducts/ZigBeeHomeAutomation.aspx.

[6] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[7] J. Augusto. Ambient intelligence: The confluence of ubiquitous/pervasive computing and artificial intelligence. *Intelligent Computing Everywhere*, pages 213–234.

[8] J. Baliosian and J. Serrat. Finite state transducers for policy evaluation and conflict resolution. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 250–259. IEEE, 2004.

[9] J. Baliosian, J. Visca, E. Grampín, L. Vidal, and M. Giachino. A rule-based distributed system for self-optimization of constrained devices. In *Integrated Network Management, 2009. IM'09. IFIP/IEEE International Symposium on*, pages 41–48. IEEE, 2009.

[10] A. Barros, M. Dumas, and P. Oaks. Standards for web service choreography and orchestration: Status and perspectives. In *Business Process Management Workshops*, pages 61–74. Springer, 2006.

[11] A. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, and M. Zorzi. Architecture and protocols for the Internet of Things: A case study. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pages 678–683. IEEE, 2010.

[12] D. J. Cook, J. C. Augusto, and V. R. Jakkula. Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing*, 5(4):277–298, 2009.

[13] A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. local computer networks. In *Annual IEEE Conference on, 0*, pages 455–462, 2004.

[14] T. Erl. *Service-oriented architecture: concepts, technology, and design*. Prentice Hall PTR, 2005.

[15] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2:115–150, May 2002.

[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, et al. *Design patterns*, volume 1. Addison-Wesley Reading, MA, 2002.

[17] S. Hadim and N. Mohamed. Middleware: Middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online*, 7(3):1–1, 2006.

[18] A. Hagedorn, D. Starobinski, and A. Trachtenberg. Rateless deluge: Over-the-air programming of wireless sensor networks using random linear codes. In *Proceedings of the 7th international conference on Information processing in sensor networks*, pages 457–466. IEEE Computer Society, 2008.

[19] J. Hauer, V. Handziski, A. Kopke, A. Willig, and A. Wolisz. A component framework for content-based publish/subscribe in sensor networks. *Wireless Sensor Networks*, pages 369–385, 2008.

[20] U. Hunkeler, H. Truong, and A. Stanford-Clark. Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, pages 791–798. IEEE.

[21] L. Mottola and G. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys*, 2010.

[22] W. Munawar, M. Alizai, O. Landsiedel, and K. Wehrle. Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–6. IEEE, 2010.

[23] C. Peltz. Web services orchestration and choreography. *Computer*, pages 46–52, 2003.

[24] A. Rezgui and M. Eltoweissy. Service-oriented sensor-actuator networks: Promises, challenges, and the road ahead. *Computer Communications*, 30(13):2627–2648, 2007.

[25] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A. Harris, and M. Zorzi. SYNAPSE: A network reprogramming protocol for wireless sensor networks using fountain codes. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON'08. 5th Annual IEEE Communications Society Conference on*, pages 188–196. IEEE, 2008.

[26] Z. Shelby. Embedded web services. *Wireless Communications, IEEE*, 17(6):52–57, 2010.

[27] Z. Shelby and C. Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley, 2010.

[28] Z. Shelby, B. Frank, and D. Sturek. Constrained application protocol (coap). *An online version is available at http://www. ietf. org/id/draft-ietf-core-coap-01. txt (08.07. 2010)*, 2010.

[29] R. Sugihara and R. Gupta. Programming models for sensor networks: A survey. *ACM Transactions on Sensor Networks (TOSN)*, 4(2):1–29, 2008.

[30] Q. Wang, Y. Zhu, and L. Cheng. Reprogramming wireless sensor networks: challenges and approaches. *Network, IEEE*, 20(3):48–55, 2006.