Mobile Networks and Applications manuscript No.
(will be inserted by the editor)

BeC^3 : Behaviour Crowd Centric Composition for IoT applications

Sylvain Cherrier · Ismail Salhi · Yacine M. Ghamri-Doudane · Stphane Lohier · Philippe Valembois

the date of receipt and acceptance should be inserted later

Abstract Service Oriented Computing (SOC) is a common way to build applications/services by composing distributed bricks of logic. Recently, the SOC paradigm has been considered for the design and implementation of Internet of Things (IoT) applications by abstracting objects as service providers or consumers. Based on this trend, we proposed in a previous work D-LITe: a lightweight RESTful virtual machine that allows ubiquitous logic description and deployment for IoT applications using Finite State Transducers (FST). Though D-LITe allows faster and more efficient application creation for heterogeneous objects, it turns out that FST design can be fastidious for inexperienced users. With that in mind, we propose in this paper BeC^3 (**Be**haviour **C**rowd **C**entric **C**omposition) an innovative crowd centric architecture, grounded on D-LITe. It provides a simpler way to compose interactions between IoT components. The idea is to reverse the bottom-up approach of SOC by a rather top-down vision in which the user expresses the expected result of his application by composing behaviours that are proposed by contributors. These behaviours are deployed on each concerned component, which then act exactly as needed to fulfil their role in the composition. The crowd-Centric aspect of this platform allows a community-based design, granting a wide panel of modular and incremental interactions for a wide variety of components. Eventually, BeC^3 will give inexperienced users the ability to organise, interconnect and compose both state of the art web-services and IoT components to create interactive 2.0-like applications for the IoT.

Keywords Services Oriented Computing; Internet of Things; Machine-to-Machine Networks;

Sylvain Cherrier · Ismail Salhi · Yacine M. Ghamri-Doudane · Stphane Lohier · Philippe Valembois
Universit Paris-Est , Laboratoire d'Informatique Gaspard-Monge (LIGM), 77454 Marne-la-Valle Cedex 2

Yacine M. Ghamri-Doudane
ENSIIE, 1 square de la rsistance, 91025 Evry Cedex

1 Introduction

The design of distributed applications on the Internet is often based on Service Oriented Computing (SOC). In such architecture, applications use logical functionalities offered by software bricks accessible via dedicated web-services. The application logic is then composed from the results of these bricks through dedicated and loosely coupled web APIs [13]. In a similar way as SOC allows the collaboration of heterogeneous web-services, our vision of the *Internet of Things* stands on the necessity to create applications from variety of available sensing and actuating technologies, whether from a hardware or a software point of view. Recent work in embedded web-service composition has also focused on SOC [14] to design IoT applications. By mimicking the role of software architects, one can imagine to build an IoT application using rich and complex composition languages like BPEL [10] and WS-CDL [17] to describe its inner interactions.

However, unlike in web APIs where web-services are installed on servers, the logic of an IoT application takes its roots from embedded WS that are deployed on various sensors and actuators. Considering that the same device can be used for different purposes, one should be able to deploy *on-the-go* (during the application execution) new bricks of logic on the device to change its utilization. This is not possible using the aforementioned business-adapted languages because they allow to compose only pre-installed static web-services. For instance,

both [14] and [20] are limited by the web services that are embedded on the devices at their deployment. After this point, any alterations of the running application might require the utilization of another WS, which can not be done without per-node reprogramming.

In this context, we defined in a previous work D-LITe [8]: a universal logic framework based on FST that allows a dynamic utilization of devices capabilities to express the different bricks of logic that may compose an IoT application. This framework introduces a hardware abstraction layer that hides the differences associated with various devices, protocols and systems. Because D-LITe is technology agnostic, it entails a new range of applications for pervasive computing allowing service creation, control and choreography among heterogeneous legacy devices. More importantly, D-LITe allows to remotely deploy versatile IoT applications by pushing bricks of logic seamlessly on running devices without per-node reprogramming.

Although D-LITe simplifies the creation process and allows less experienced users to compose embedded web-services, it still involves the design of FST and the definition of exchanged messages, which is not trivial. We believe this complexity in the application creation process is one of the last obstacles that prevent the Internet of Things from its awaited democratization. With that in mind, this paper describes BeC^3 (**Be**haviour **C**rowd **C**entric **C**omposition), an innovative framework that brings the flexibility of D-LITe with the benefits of crowd-centric architectures to allow users to easily design IoT applications. Using BeC^3 for an IoT application creation is less complex. Our approach is to reverse the composition process when the implementation is entrusted to an experimented software architect. In fact, BeC^3 dismisses this role of architect by providing for each device generic bricks of logic implemented in a repository by a savvy developers community. The input(s) and output(s) of these bricks are standardized and typed to allow the user to be the *real time* architect of its application. One can, thus, be certain to create compatible and meaningful interactions between various devices without any technical knowledge about how bricks of logic are implemented or how messages that drive the application are exchanged.

The idea of offering a large panel of ready-to-use pre-written software components makes sense because it implies an iterative growth of the available bricks of logic for the final users. According to the 1%/9%/90% Crowd Centric organization presented in [5], we propose that 1% of BeC^3 users (*system builders* which requires significant technical knowledge) implement the D-LITe logical interpreter on existing devices. Then 9 % of BeC^3 users (*services programmers*) develop software components (bricks of logic) using D-LITe and add them to the central repository. Finally, inexperienced users (the remaining 90%) can compose available compatible components, and create their own applications that retrieve contents autonomously in the physical world.

The rest of the paper is organized as follows: Sections II and III discuss the background and related work of service oriented computing and the prior effort in adapting it to WSANs and the IoT. Section IV defines the context and the motivation that lead us to the main concepts of BeC^3 and details the model behind it and its major features. In Sections V and VI, we illustrate the practical benefits of our solution through an illustrative scenario that presents our implementation of BeC^3 and its potential benefits in real IoT applications. Finally, Section VII concludes the paper.

2 Related Work

Since the mid-nineties[18], researchers and industrials have mainly dealt with primary issues inherent to the constrained nature of sensing and actuating networks. Their work often concerns hardware optimization, energy consumption, communication reliability and deployment rationalization. But for almost a decade now, the Internet community is providing interesting solutions in bridging the gap between isolated WSAN and the World Wide Web.

Building an Internet of Things (IoT) that links technologies such as WSAN, networked embedded devices and Internet infrastructure is the goal of many projects. The authors in [11] proposed Contiki, a pioneering operating system whose aim is to bring IPv6 connectivity directly to constrained devices [12], and allow their large scale interconnection in a Wireless Embedded Internet. This initiative gave birth to the IETF standard 6LowPAN [23] and hereafter to a series of innovative higher layers protocols such as CoAP [24], Observe[1] and Link Format[2]. Thus, some other projects work on the interaction of all IoT devices. Linked to the Future Internet programme, FI-Ware [1] is an European project to "*build a service infrastructure*" for "*developping Future internet Applications in multiple sectors*". The IoT part of this project is IoT6 [27]. IoT6 uses 6LowPAN but is more global. It aims to offer services discovery in order to realise a full integration of all technologies and devices. Octopus [3] is another project that "*integrate[s] and coordinate[s] heterogeneous devices and systems*" while being "*pervasive, [...] permitting the seamless in-*

[1] http://tools.ietf.org/html/draft-ietf-core-observe-03
[2] http://tools.ietf.org/html/draft-ietf-core-link-format-14

tegration" of all devices. We can also cite SENSEI [21] that "*plays a leading role* [...] *to create and underlying architecture and services for the future Internet*" that will "*connect the physical into the digital world*". This project aims to "*define an architecture that* [...] *deal*[s] *with large number of globally distributed WSANs and interoperability of heterogeneous devices and platforms*".

In their extensive survey [2], Atzori *et al.* address this new trend and present its most recent literature. They illustrate how this new paradigm mixes the concepts of Internet with WSAN usages for the creation of new applications that dig their content from Web Enabled objects. Using concepts from the SOC paradigm, solutions have been proposed to compose applications for the IoT just as one would compose state-of-the-art web-services in software engineering. In [13] for instance, a central point collects, uses, mixes and computes data or send orders to distant nodes.

In constraint network such as WSAN, energy is an important issue, so limiting the transmission is always a good idea to increase device lifetime. Programmers use solution to do the work in place, by using devices processing capability. Abstract Region [26] is an example of solution that tries to limit network usage. Abstract regions are build depending on application needs in order to locally compute data. Authors claim that "*it is generally desirable to perform local compression, aggregation, or summarization within the sensor network to reduce overall communication overheads*". However, this solution is limited to WSAN with multiple units of a restricted set of devices, while our vision of IoT is based on a large set of unique devices offering very different services.

In fact, both centralized and decentralized solutions exist in the literature to initiate interactions between distant objects over the Internet. Known respectively as Service *Orchestration* and *Choreography*, such approaches offer access to data through embedded web-services but in two different ways. In a previous work [9], we demonstrated analytically and empirically that choreography offers better performances, reliability and energy efficiency on constrained networks. Furthermore, recent studies on the subject tend to show the impact of service choreography on the way web applications are deployed and maintained. In their position paper, the authors of [25] argue that "*service design needs two or more abstraction models*" and insist on the importance of choreography as a fundamental design principle. We argue that this theory of web choreographies is a major step forward in web applications. It emphasises the gap between what is expected at the global level (the application choreography) and what happens practically (the code implementation). This, we believe, is particularly pertinent for automating web- enabled objects in the Internet of Things. Indeed, if we abstract the devices' internal implementation, an IoT application is nothing more than a set of various messages exchanged between logical bricks (devices or software) that convey its semantic to the system.

The idea of needs projection that is introduced in [6] can thus be exploited to ensure more versatility during the design process, allowing the deployment of new bricks of logic during the execution. Authors of [22] investigate the utilization of web-service choreography as a way to compose distributed applications. They propose a theoretical framework that studies its impact on service interactions and establish a set of rules that formalize the exchanges between nodes and the implementation of local web services. However, in order to resolve issues related to asynchronous services execution, the authors define the notion of dominant/dominated services, which eventually induces a form of orchestrated organization that is inconsistent with our choreography philosophy.

Macro programming is also an interesting idea because it offers a simple and high-level solution to quickly create applications. The architecture of IoT is often based on multiple devices that embed different level of processing capabilities. Macro programming in WSAN gives an abstract view of stakeholders and hides hardware specificities. For example, Kairos [15] "*presents an abstraction of a sensor network as a collection of nodes that can all be tasked together simultaneously within a single program*". With Kairos, the programmer uses "*a shared-memory based parallel programming model*" of "*loosely synchronizes*" nodes in order to respect the WSAN constraints. Kairos nodes can share a remote data access across the network. But even if the synchronization is lazy, we believe that our event-centric solution provides better results. BeC^3 exchanges semantic messages and not the data that are used to obtain these semantic messages. Moreover, Kairos and [26] are for large set of identical devices while we want to build IoT applications that deals with a wide variety of nodes.

Another way to compose web-services is based on the *Roman Model* introduced in [7]. The principle here is to characterize services by "*their conversational behaviours compactly represented as finite transition systems*". Because of its transitional view of the system, the Roman model is commonly used to abstract exchanges between its interacting elements. [4] for instance exploits this principle to design a new technique that quantifies optimal composition possibilities that could eventually allow a just-in-time deployment of applications. Our goal in this paper is to allow an even

more intrusive approach by imposing on-the-go specific bricks of logic (services) on nodes and not only deal with their embedded static services. Based on an abstraction of the conversations among IoT components and the FST representation of their respective "Behaviour", we propose to exploit the innovative concepts of the Choreographies Theory and the transitional properties of FST modelling to create IoT applications. In fact, our vision is to consider its design as a top-down process rather than a bottom-up approach in which an Architect can only combine existing services that are statically deployed on devices.

In their reference paper [14] about the Web Of Things (WoT), the authors plead for a "*user-friendly representation*" of objects. They argue that the "*Web-enablement of smart things delivers more flexibility and customization possibilities for end-users*". They investigate the utilization of Resource Oriented Architecture (ROA) for constrained devices, and present as a result a Web Mashup (an application for building "*opportunistic integration*" of appliances) that illustrates how a WoT composition can be performed using simple RESTFul APIs. This illustrative example is interesting because it provides a first attempt to create interactions between various objects and logical components using an automated editor[3] that allows "*to visually create Web mashups by connecting building blocks of resources*".

Nevertheless, [14] remains a relatively complex solution that requires a certain knowledge about the devices and the logical components necessary to design an application. More importantly, the static nature of the services offered on each device restrict severely the versatility of the applications. Indeed, each alteration of device individual functionalities requires a physical access, and the development of a new program that will run on it. And even when the object functionalities cover the user's needs, the creation of the application may still be complex. For instance, designing a simple heating control with *ClickScript* requires to understand that a temperature sensor can send values, that the combination of them with a ">" object and a given value (e.g. 20) allows to send a boolean activation to the heating system. Not only this can be conceptually out of reach for the lambda end-user, but also resource-consuming because the decision (if temperature> 20) is taken at a central point, as the sensor is either constantly pushing data or regularly requested by a centralized controller.

Our work goes further than existing RESTful approaches. BeC^3 allows to increase the logic by running a maximum of processing on the devices to limit transfers and energy depletions. If we consider the previous example, a better solution would have been to make the temperature sensor send only one specific message directly to the heating system whenever the defined threshold (> 20) is reached. In classic RESTful composition schemes, if this threshold message is not offered natively by sensor hardware, it can only be done by a specific program that has to be uploaded on the sensor itself. Using BeC^3, one could push this new feature directly on the desired node(s) seamlessly by behaviour remote deployment. In addition, this paper describes a design method in which services are composed in a more semantic way (*describing a goal*) rather than typically functional (*giving a value*). This *Event-Centric* vision (instead of *Data-Centric*) is central to our distributed approach. It shows that the more nodes are autonomous, the better it is for constrained networks and consequently for the whole reactivity.

[3] ClickScript - http://www.clickscript.ch/ a Mashup javascript tool to interconnect virtual objects

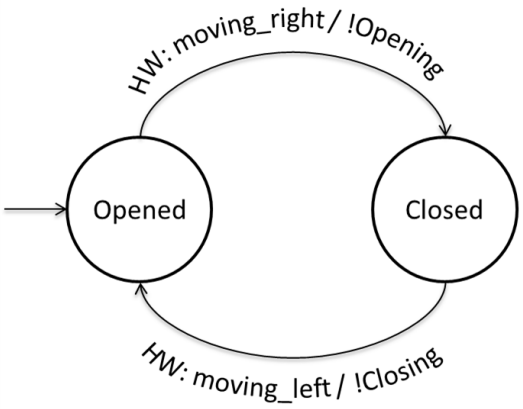


Fig. 1 An example showing how a FST can represent a "smart door". A hardware message (HW) triggers a state change and sends a message to listening objects.

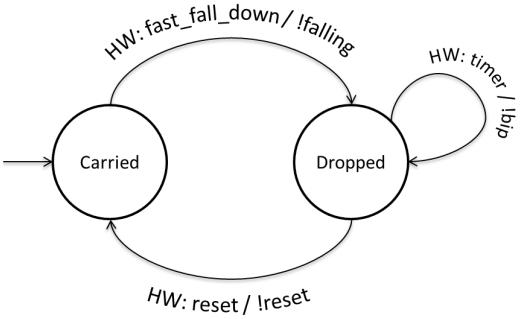


Fig. 2 The same motion sensor has a new semantic role just by changing its FST.

3 Background

This section describes briefly D-LITe, the lightweight RESTful virtual machine on which BeC^3 is grounded.

This abstraction layer offers a universal access and representation to the native functionalities of various devices regardless of their technological characteristics. D-LITe standardizes the description of a device functionality using *Transducers*. Transducers are a derivative form of FST with an additional *output alphabet*. In our proposition, one could understand a FST as the piece of code that ties up the devices native hardware functionalities with each one of its potential semantic utilizations (behaviours). For instance, a motion sensor (accelerometer) can have several FST depending on its purpose: If this sensor is put on a door, a FST could describe how it might detect when the door is opened or closed (Fig 1). But if the same sensor is put with a set of keys, another FST could describe how it can send notifications whenever they are dropped somewhere (Fig 2).

In [8], we detail how a simple REST messaging system can interconnect a large scale of *D-LITeful* devices, and allows to express and deploy the rules that will build an *Internetable* choreography. After deployment of these rules, every device functionality could be considered as a common web-service that takes part in the execution of an IoT application.

The reason behind the utilization of such transitional representation is, first, to cope with IoT components heterogeneity (sensors, actuators, web servers, etc.) and to relieve them from any language/operating system dependencies. Furthermore, this choice allows a dynamic on-the-go deployment of new applications throughout very concise rules without physical intervention on the concerned nodes. Thus, by invoking dedicated web-services offered on each node by D-LITe, one can simply deploy simple bricks of logic to serve a larger web choreography.

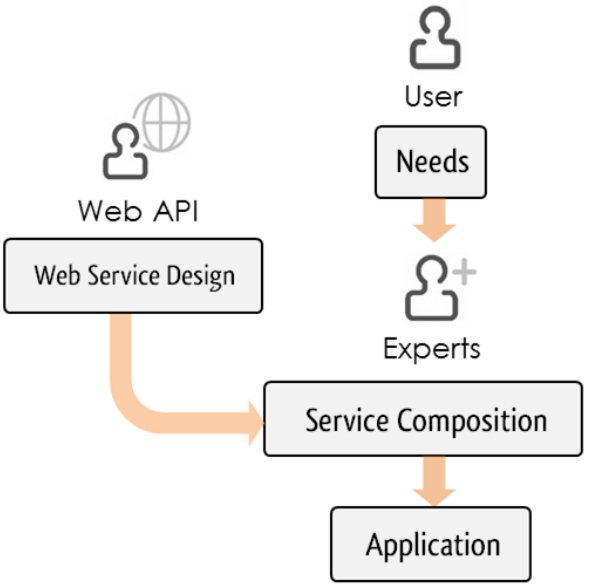


Fig. 3 Usual SOC Design: A user expresses his needs to an Architect. The Architect builds the corresponding application using web-services offered by others programmers.

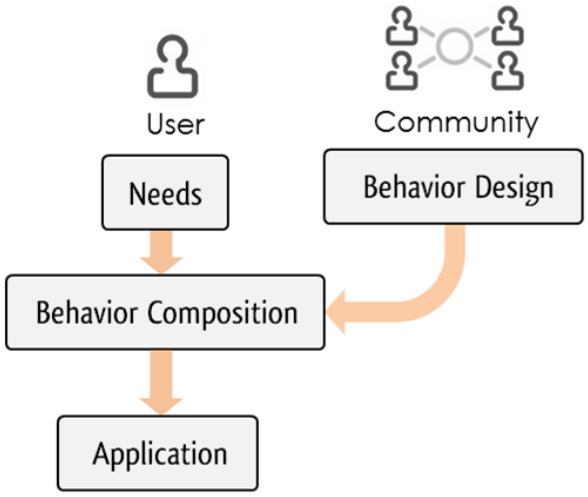


Fig. 4 In our solution BeC^3, a User directly solves his needs by composing *Behaviours* offered by a community of programmers. Each *Behaviour* is then installed on the corresponding node.

4 Crowd Centric Service Choreography

4.1 Motivation and Key Idea

In common web engineering, two profiles of specialists are involved in the application creation process: First, programmers whom use programming languages (eg. C, Java, Python, etc.) to create and expose web-services, then, web architects whom have recourse (only thereafter) to workflow languages such as BPEL or WS-CDL to orchestrate these services (Fig 3) from a central point. Such strong chronological/technical dependency between both profiles implies necessarily the cohabitation of distinct skill levels. This complicates the development curve of web applications, a fortiori in the web of things where this separation is less justified. Indeed, the elements involved in the application belong mostly to one end-user and not to distinct corporations. In fact, services that are embedded on sensors or actuators and the application that controls them are often destined to their owner. The services composition should thus be much more agile and dynamic to allow appliances to be programmed by and for the user.

In this paper, we believe that the user needs to be both the developer and the architect of its own IoT applications. Yet, it seems unreasonable to expect him to master such complex and different skills as web development and work-flow management. That is the main reason behind our choice to crowd-source the applica-

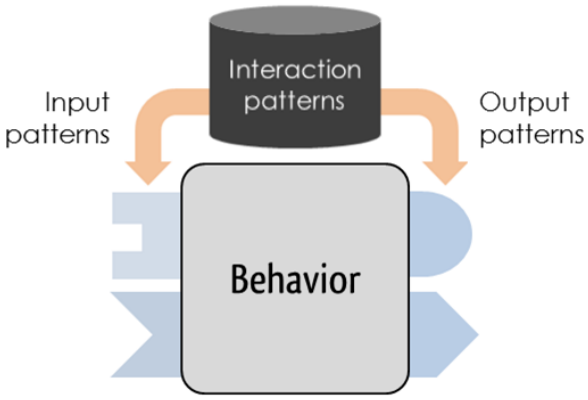


Fig. 5 A *Behaviour* in BeC^3 is characterised by the way it interacts with others. *Interaction Patterns* describe typical interactions, and *Behaviours* inputs/output are restricted to *Interaction Patterns*.

tion creation process in the IoT. Moreover, the user's requirements can change rapidly depending on its context and daily needs. This may require its applications (and thus services) to be frequently and promptly reconfigured and re-organised. Our notion of crowd-sourced behaviours combined with the agility offered by D-LITe would grant this flexibility in the application deployment and execution allowing the user to control constantly and remotely its IoT devices and services.

As a matter of fact, we argue that the complexity of existing service composition techniques can be transcended using BeC^3 because in opposition to common enterprise processes, most logical elements of the IoT are fairly straightforward. We noticed that interactions between IoT components can be easily recognisable and classified (activation / deactivation, enslavement, transmission, etc.). It becomes thus possible to characterise their logic and normalize their exchanges with one another. This classification is the hallmark of BeC^3. Our system allows to feed the community with a set of basic tools and a common language that will eventually help them to create prefabricated bricks of logic that could be easily assembled in one IoT distributed application.

4.2 Modelling assumptions

BeC^3 reverses the methodology used in common service composition. Instead of composing static web-services, elaborated bricks of logic are pushed directly on nodes, making the deployment of new IoT applications seamless because D-LITe nodes becomes remotely programmable. This flexibility allows to deploy different semantic interpretations, that we call *Behaviours*, to the device native functionalities depending on its real life utilization. Fig 4 shows how these behaviours, when installed on a device, can help final users to easily compose their own IoT application. The careful reader will notice that BeC^3 focuses on the interactions between objects and their direct interlocutors. The idea here is to pair, for each device, its behaviour (brick of logic) with a set of interaction patterns that it is able to process.

To allow an incremental growth of available behaviours, the BeC^3 collaborative platform is based on a model composed by three entities: **nodes**, the **behaviours repository**, and **users**.

- First, **nodes** (whether devices or web-services) must implement a virtual-machine-like framework (D-LITe [8]) to support the deployment of distributed logic and publish-subscribe capabilities over the Internet. This allows to easily deploy a service choreography interconnecting distant devices and services using finite state transducers.
- Then, and as depicted in Fig 6, the **repository** enlists the available Behaviours (FSTs) that eventually will run on D-LITeful nodes. At the reception of expected messages, each concerned node reacts depending on its behaviour, then triggers the transmission of a set of messages to its interlocutors (other behaviours). These messages comply with what we define as "*Interaction Patterns*" (Fig 5). Wherein, an interaction pattern is a formalization that standardizes the input(s) and output(s) of each device to ensure its interoperability.
- Finally, different profiles of **users** play specific roles in the application creation process. Indeed, BeC^3 relies on the '*participation inequality*' [19] that describes the *01/09/90%* rule.

This Crowd-Centric approach has been used to solve complex problems and is known for its efficiency in heterogeneous systems [5]. A common way to use its principle is to allow 90% of the system's users to consume the available resources, 9% to provide assistance, and 1% to do the heavy work by designing and maintaining the collaboration platform. In BeC^3, the 90% are users who want to create and deploy their own Internet of Thing applications with no required programming skills. The 9% create *Behaviours* that run on a specific object category. They are in fact FSTs that provide semantically "meaningful" usages for the 90%, and comply with *Interaction Patterns* (Fig 6) to allow a maximum device interoperability. The last 1% take care of implementing D-LITe on legacy devices such as sensors, phones and appliances. They may also punctually define new *Interaction pattern* (Fig 6). Thanks to the inherent modularity of FSTs and the availability in a "public" repository of several devices usages, it becomes easier to the community to create and assemble bricks

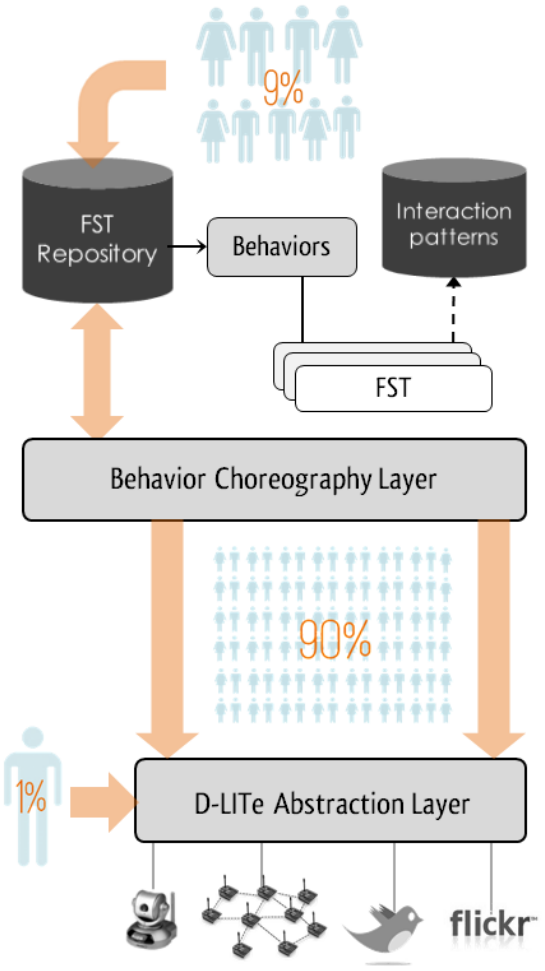


Fig. 6 A user selects a BeC^3 *Behaviour* for each of its own objects. The logic is compatible because Input/Output implements the same BeC^3 *Interaction Pattern* (*Boolean Interaction* for example). These *Behaviour* are written by some other BeC^3 users.

of logic and IoT applications in a Web 2.0-like fashion. The 9% share their behaviours by pushing them in the repository while the 90% simply use them by a seamless remote deployment.

4.3 Composition Model

Because BeC^3 aims to simplify the service composition of an IoT application for lambda users, its model is based on two clear notions : Behaviours and Interaction Patterns. The class diagram (Fig 7) describes the organisation and the links between different elements of BeC^3. For each object or device, it exists a list of available behaviours in the repository. Each behaviour may have one or more interaction patterns for both its inputs and outputs.

4.3.1 Interaction Patterns

Creating a distributed IoT application in BeC^3 involves the definition of actions/reactions for all its collaborating elements, as well as the exchanged messages between them. We propose in this work a first classification of these exchanged messages using what we define as *Interaction Patterns*. An interaction pattern, when affected to a certain behaviour (see below), ensures its compatibility with other elements by listing the types of its output(s)/input(s). This allows to simply verify if its composition with other behaviours makes sense in terms of compatibility. A given cardinality expresses constraints about the required number of received/transmitted messages of a behaviour. We identified an initial list of interaction patterns that normalize exchanges between behaviours:

1. **Boolean Interaction**: The most obvious pattern which allows activation and deactivation. In this case, two messages are exchanged, the first (*on*) activates and the second (*off*) deactivates.
2. **Bounded Counter**: This pattern helps to increasing or decreasing a level or a value. It can be used to design sliders and dimmers for example. 4 different messages are needed to define gradual progressions: *up* and *down* act relatively to the current value. *Off* and *Full* indicates the absolute minimum and maximum values. This IP can be used to control lighting, sound volume or a camera zoom.
3. **Coordinates**: Used to provide gliders/drivers, it enables to position objects by exchanging *north, south, west* and *east* messages. A joystik, or even a mouse can offer this pattern as output for example, while a motorised camera can require it as input.
4. **Toggle**: This interaction patterns acts as a flag (either up or down). Each time the state of the flag changes, a single message *toggle* is transmitted. This interaction, though very simple and semantically poorer than the *Boolean Interaction*, can be useful to command simple alarm buttons for example.
5. **Send**: Needed to transmit content (given as a parameter in the message) to other behaviours. Such interaction pattern is very useful for content-centric interactions particularly with classic web-services. Possible utilizations could allow to interconnect devices with micro-blogging services or to transmit multimedia content for instance. It issues one message (*send*) that includes the desired content (text, binaries, streams, etc.), e.g. *send(Full capacity reached)*.
6. **Notification**: Used to notify other behaviours about a specific change in the device's state by transmitting a single message (*notify(msg)*). This IP may

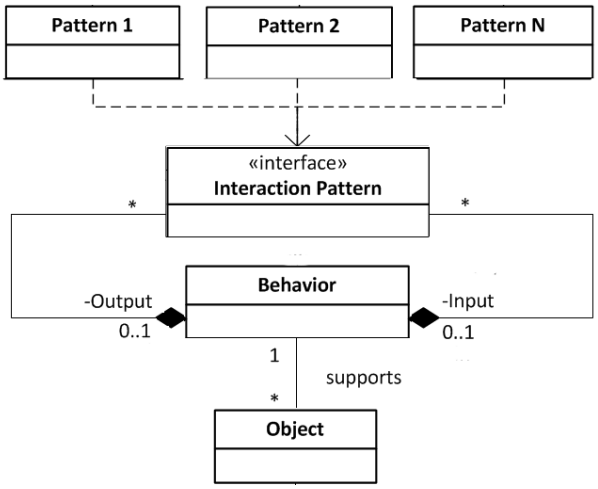


Fig. 7 This Class Diagram shows how each *Behaviour* is linked to a specific object, and offers or requires some *Interaction Patterns*.

seem similar to the **Send** pattern. However, when receiving a notification in this case, a behaviour will surely go in a particular state, while the main purpose of **Send** is to transmit a content without necessarily changing the state of the device. The "state changing" notion is important here because it changes the semantic utilization of such IP. The Notification IP can be used to issue alarms or alerts such as *notify(Fire)*. When receiving such a message, a device could move to a special inactive state, while the transported message (*Fire*) can be ignored.

The first version of this standardized *Interaction Pattern* list can already characterise a large choice of *Behaviours*, offering BeC^3 users the ability to build a large panel of IoT applications.

4.3.2 Behaviours

They are the logic units presented in the repository and ready to be deployed by the 90% without physical access to their devices. They are small pieces of program that react to external stimuli (from other behaviours) or internal ones (from their own hardware). Behaviours are expressed as Transducers [8], where each state transition can cause the transmission of messages to their own hardware (sensors, processor, actuators, etc.) or to other behaviours (devices or web services). One important contribution of this work lies in the formalization and characterisation of exchanged messages between behaviours using interaction patterns. An arbitrary behaviour is said to be compatible with one or more of its peers if they all share a set of compliant input/output interaction patterns (Fig 7). Note that each behaviour has a set of input and output Interaction Patterns (IPs) that are either required or optional. Thus, it can only send and receive messages according to its set of IPs.

This restriction allows other peers to communicate with it while ensuring that each transmitted message is to be understood by its destined behaviour (triggers a transition in it). Indeed, since BeC^3 emphasises the interaction and not the object which runs the behaviour, it becomes possible to use any kind of objects as long as they have the corresponding interaction patterns. The behaviour description provided by the 9% is expressed via a XML file (Fig 8) that describes the rules of the transducer, the type of devices on which it runs and the cardinality of each one of its input (*understanding*) and output (*talking*) Interaction Patterns. This cardinality is represented in our notation by the number of inputs (respectively outputs) that are required for a described behaviour, followed by an optional '*extend*' attribute that indicate if the value is a minimum.

For instance, an input (*0 extend*) indicates that the Interaction Pattern is optional, and thus that the concerned behaviour can understand it and act accordingly, but it is not mandatory. A *0 extend* cardinality can be used to express that the IP **Alarm** can be understood by the behaviour without being required. Thus, the reception of an **Alarm** message will trigger an action of the behaviour that will put it in a specific state (defined by its designer). A cardinality of 1 implies that the concerned IP is required. It is often used to establish a master/slave interaction between two devices. A typical example is a lamp controlled by a button. Here, the interaction pattern can be **toggle** in the input of the lamp and the output of the button, and its cardinality set to 1. A cardinality of N is used to describe an interaction with N different devices. Such case can be used to describe a behaviour that needs a specific number of inputs before launching an action. Finally, adding the *extend* attribute adds the notion of "at least" to the number of interacting devices. A 3 *extend* would, for example, indicate that three or more behaviours are needed to trigger an action. Hence, we obtain a correlation between the cardinality of interaction patterns and N ($N >= 0$): the number of interacting behaviours, where a cardinality of:

1. **N** describes a behaviour that requires messages from N and only N interaction pattern compliant device(s),
2. **N extend** describes a behaviour that could accept messages from at least N devices.

Note that in the XML description (Fig 8) of a given behaviour, the cardinalities are indicated for each one of its interaction patterns, whether as an input (*understanding*) or an output (*talking*).

```
<?xml version="1.0"?>
<behaviour>
 <fst id='XXXX' />
 <object id='YYYY' />
 <description>
  This FST is referenced under number XXXX.
  It runs on object YYY
 </description>
 <understanding>
  <IP id='ipX'>0+</IP>
  <IP id='ipY'>5+</IP>
       .....
 </understanding>
 <talking>
  <IP id='ipZ'>1</IP>
 </talking>
 <fst> rule1 rule2 rule3 ....</fst>
</behaviour>
```

Fig. 8 BeC^3 XML description of a *Behaviour*

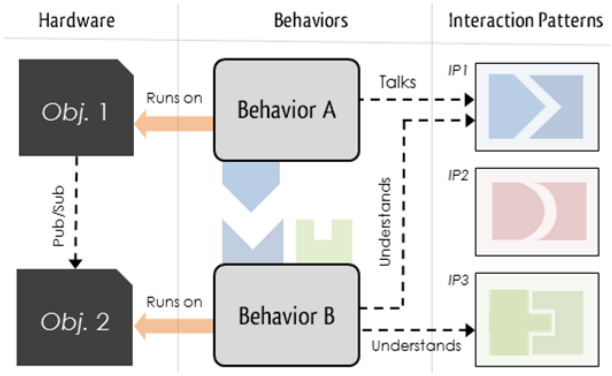


Fig. 9 Two Objects running BeC^3 *Behaviours*. They interact because the two *Behaviours* correctly match the required *Interaction Patterns*. Object 2 runs a *Behaviour* that "understands" IP1 (needs it as an input), and optionally IP3. Object 1 runs a *Behaviour* that "talks" IP1. So publish/subscribe can be done.

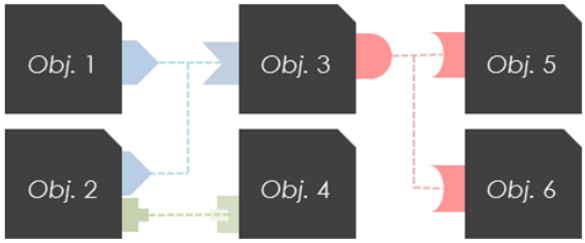


Fig. 10 A BeC^3 application where multiple objects interact. If outputs match correctly needs and conversely, the Choreography may be deployed and started.

4.3.3 Deployment

BeC^3 is based on an **a posteriori** deployment of the necessary behaviours to the service choreography. An *Internet of Things* application may involve personal elements (devices, smart objects, etc.) and common services offered on the Internet (distant sensors, web-services, etc.). Because he is the owner of different objects used in its application, a user has the ability to automatically configure them when and as needed. The first contribution of D-LITe [8] offers the possibility to extract the diversity and the complexity of existing programming tools to provide a universal platform for writing logic elements constituting a distributed application.

BeC^3 goes even further because it provides the possibility to choose among a set of pre-written components and deploy them on different devices, while ensuring their behavioural compatibility (Fig 9). Hence, to achieve its Mash-up (the combination of the different behaviours involved in the application), a user might use BeC^3 mashup tool which identifies the type of devices on its private network during a discovery phase. To this inventory, the user can add other web connected components/services, either via state-of-the-art web technologies (email, micro-blogging, web-services, etc.) or other web-enabled devices (public sensors, actuators, etc.).

For this set of *D-LITeful* components, a list of compatible behaviours is generated from the BeC^3 *repository*. The user can then select the desired behaviour to be deployed on each one of its components according to a brief description of the behaviour's functionality. After selecting a behaviour for each component, the user can use the mashup tool to link its devices inputs and outputs to one another depending on its application purpose. Finally, because the links between objects are the constituent elements of the IoT application (Fig 10), BeC^3 verifies the interaction pattern compatibility between all linked components using the coherence checking mechanism described in the following subsection.

4.4 BeC^3 Coherence Model

The BeC^3 formalization brings out the necessity to verify the coherence of an application composition. Indeed, the existence of interaction patterns allows the utilization of a formal scheme that verifies if the deployed choreography does not include aberrations that could prevent it from a flawless execution. For example, linking a sensor that has an output Toggle IP with an actuator that needs a different input IP may cause the failure of the whole choreography execution. This section details a mechanism that could prevent such inconsistencies using a simple model that verifies for a given choreography if all the behaviours are compatible and thus capable of executing a the distributed application.

Consider a choreography C as $C = \{O, B, P\}$ where:

- O is the set of involved Objects,
- B the Behaviours set to be deployed on Objects,
- P their links (publish-subscribe relations).

If we focus on an element C_x with $0 < x < n+1$ and $n = |O|$, we have : $C_x = \{o, \beta, A_x\}$. With $o \in O$ the object running the behaviour$\beta \in B$ and $A_x (A_x \subseteq O)$ the set of objects listening to o.

With that in mind, we may also state that an object o has a *preceding* list of objects O^- and a *following* list of objects O^+. Preceding objects talk (*send messages*) to o, while following ones listen (*expect messages*) to it. Thereby, an object o is *preceded* by another object o' iff $o' \in O^-$ and is *followed* by an object o'' iff $o'' \in O^+$, thus we have $O^- \longrightarrow o \longrightarrow O^+$.
Moreover, for each behaviour $\beta \in B$ we specify interaction patterns I according to the following constraints:

- β has a number of required interaction patterns for both its inputs I_{in}^{req} and outputs I_{out}^{req}. We formalize these lists of interaction patterns using multisets since a behaviour may require more than once the same interaction pattern as input or output. Indeed, multisets, in opposition to common sets, can illustrate this interaction pattern redundancy,
- β has two sets of optional interaction patterns I_{in}^{opt} and I_{out}^{opt} (denoted by the cardinality and the sign +). We use here a simple set (no redundancy), because we focus only on the optionality of the interaction pattern (their number is not important)

Therefore a behaviour β handles 2 sets and 2 multisets of interaction patterns as follows:

$$I_{in}^{opt} = \{ip_0, ..., ip_n\}, I_{in}^{req} = \{\{ip_0, ..., ip_{n'}\}\}$$
$$I_{out}^{opt} = \{ip_0, ..., ip_{n''}\}, I_{out}^{req} = \{\{ip_0, ..., ip_{n'''}\}\}$$

Considering that:

- in C, each tuple C_x means that an object $o = O_x$ is running a behaviour $\beta = B_x$,
- β uses referenced interaction patterns,
- an object o has preceding O^- and following objects O^+ that respectively listen and talk to it,

we can build the whole list of Outgoing Interaction Patterns OIP in the objects set O^+ built out of two elements: $OIP = \{OIP^{opt}, OIP^{req}\}$ with

- $OIP^{opt} = \{ip_0, ..., ip_n\}$ its set of optional IPs,
- $OIP^{req} = \{\{ip_0, ..., ip_{n'}\}\}$ the multiset of required ones.

Building the Optional Outgoing Interaction Patterns set OIP^{opt} for an arbitrary object o follows this rule :

$$OIP^{opt} = O_0(I_{in}^{opt}) \cup O_1(I_{in}^{opt}) \cup \ldots \cup O_n(I_{in}^{opt})$$

with $O^+ = \{O_0, O_1, \ldots, O_n\}$ (Note that any multi-occurrence of the same Interaction pattern is removed by the union of $O_n(I_{in}^{opt})$).
Regarding the input and output multi-sets of required interaction patterns of o, the exact number of IPs issued for (or requested by) other objects must be expressed. We use multiset sums (not unions) as follow : $OIP^{req} = O_0(I_{in}^{req}) \oplus O_1(I_{in}^{req}) \oplus \ldots \oplus O_n(I_{in}^{req})$.
OIP enlists the expected IP by the set of following objects O^+. Using the same logic, building the list of Incoming Interaction Patterns $IIP = \{IIP^{opt}, IIP^{req}\}$ is expressed by an equivalent formulation using the output Interaction Patterns sets of all objects in O^-.
Building IIP and OIP allows to perform consistency checking of the application to be deployed on the devices. Indeed, if the behaviours and the publish-subscribe links selected by the user are not positively verified, it means that the constraints imposed by the behaviours are not met. Hence, for each object o running a behaviour β, we have:
$IIP \longrightarrow o \longrightarrow OIP$ giving $IIP \longrightarrow \beta \longrightarrow OIP$
Replacing IIP and OIP by their respective content:

$$\{IIP^{opt}, IIP^{req}\} \longrightarrow \beta \longrightarrow \{OIP^{opt}, OIP^{req}\}$$

while behaviour β contains 2 sets and 2 multisets:

$$\{IIP^{opt}, IIP^{req}\} \longrightarrow \{I_{in}^{opt}, I_{in}^{req}\}$$
$$\{I_{out}^{opt}, I_{out}^{req}\} \longrightarrow \{OIP^{opt}, OIP^{req}\}$$

Considering this model, the BeC^3 tool which is responsible of enabling the composition and its deployment can check the validity of the choreography using the following rules:

1. $IIP^{req} \subseteq (IP_{in}^{opt} \oplus I_{in}^{req})$: All required interaction patterns in O^- must be also in the list of IPs understood by the object itself.
2. $I_{in}^{req} \subseteq (IIP^{opt} \oplus IIP^{req})$: All required Interaction Patterns of the object must be feed by O^-.
3. The same reasoning is applied to the object outputs, and must verify that $OIP^{req} \subseteq (I_{out}^{opt} \oplus I_{out}^{req})$ and that $I_{out}^{req} \subseteq (OIP^{req} \oplus OIP^{opt})$.

If all the constraints expressed by behaviours are satisfied and verified, BeC^3 proposes to deploy the selected choice on all nodes. Otherwise, the application as composed by the user is not valid, and can not be deployed until the conditions are met. For *D-LITeful* nodes, the deployment mechanism is described in [8]. The transducer representations of behaviours are described using our language (SALT [8]), a simple HTTP messaging allows to configure the services installed on each *D-LITeful* node. This transducer code is available in the XML file (Table 8).

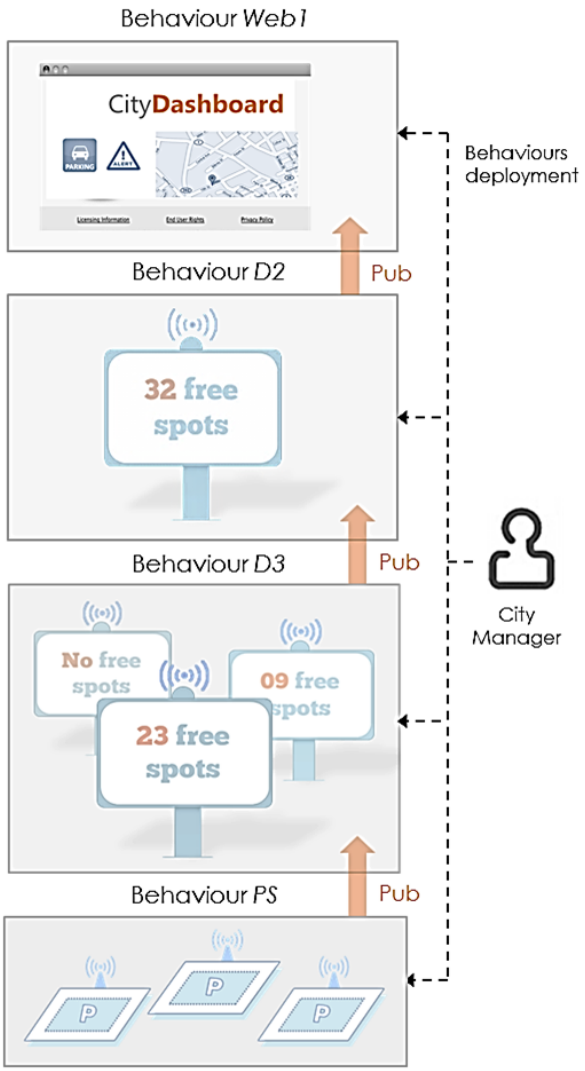


Fig. 11 In a Smart City, the manager deploys each chosen *Behaviours* on corresponding object in order to build a parking spots monitoring application.

Table 1 A Smart City *Behaviours* repository

Behaviour	understands	talks	description
Parking Spot			
PS1 : Std	-	Notif.	send on or off if the place is free or not
PS2 : countingFree	-	Bounded C. (1,n)	Increase or decrease the number of free places
Display			
D1 : std	Bounded C. (1,n)	-	Count information and display total
D2 : std with alarm	Notif.(0,n) Bounded C. (1,n)	-	As D1, but turns to red if receiving an Alarm
D3 : std, alarm and cascade	Notif.(0,n) Bounded C. (1,n)	Slider(1,n)	As D2, but send count information in cascade to another Slider
D4 : Global counter and send	Bounded C. (1,n)	Send(1,n)	As D2, but send Total obtained to another(s) Object(s)
Virtual Object			
Web1 : Push data to Web	Send(1,n)	-	Update Free Parking lot number on the Town Website

5 Illustrative Application - Town Automation

5.1 Initial scenario

Smart Cities [16] are a good example of what IoT applications can provide. As part of a Smart City project, the City's information system Manager has access to a set of objects (Fig 11), such as the city website (through a BeC^3 virtual object giving access to website's content), many display panels located in various parts of the city and several sensors available on each parking spot (in charge of detecting a car presence). All these objects are *D-LITeful*, and can potentially communicate with each other. The different behaviours and constraints for all these components are presented in Table 1. The car detection sensors send information to the display units to indicate parking spots availability. To process the number of available spots, the behaviour PS2 (Table 1) must be installed on each sensor. The behaviour D1 has to be deployed on the display units to receive information from sensors and display the results. To ensure compatibility, the PS2 behaviour has an output *Bounded Counter* interaction pattern, while D1 has the *Bounded Counter* as an input.

5.2 Evolution of the initial scenario

Thanks to the flexibility of our architecture, a richer application can be easily deployed without physical access to existing devices: The system manager can add *D-LITeFul* smoke sensors in underground car parks, and display units in a new area of the city. The new display panels can still indicate the number of remaining spots, but also display fire alerts in case of smoke detection. In this scenario, the devices are organized in cascade (Fig 11) i.e., sensors publish their messages to each area display unit to process the sum of free spots thanks to the Bounded Counter interaction pattern. Every progress of this sum affects the evolution of the main display unit which is also running a behaviour implementing Bounded Counter. The main display unit communicates every change to the website via the virtual object in charge of web access. The behaviour of each display panel implements the proper output for this communication. Finally, by choosing this time the

behaviour D3 on each display unit, we can create an alarm system that will relay automatically alerts on all the panels. To sum up, the manager needs to :

- deploy *PS2* on each car detection sensor,
- deploy *D3* on all display units and subscribe them to all the car detection and smoke sensors,
- deploy *D4* on the main display, and subscribe it to each display panel in order to be aware of any variation of available spots,
- deploy *Web1* on a Virtual Object responsible for linking the website to the other devices. It has all the appropriate access for real-time modification of the website content, providing dynamic display of the number of remaining spots. This object is subscribed to the main display unit.

Note that each algorithm implemented in a behaviour can be enhanced with new processing capabilities and/or exchange opportunities via different interaction pattern. The BeC^3 repository allows all contributors to submit new behaviours or improve older ones.

5.3 Bec^3 limitations

BeC^3 handles a very small vocabulary for the messages generation. This is a very strong constraint for the Behaviours designer. He is limited to these few words for characterizing the inputs and outputs of the logic that is being described. This limitation allows BeC^3 to make better compatibility checks of the Behaviours that are used in an application. Moreover, we have chosen not to include the sender identification in the exchanged messages because we think that this gives BeC^3 an important genericity and scalability. But theses approaches may lead to difficulties in the FST writing. For example, we can add two new sensors that detect if parking doors are closed or opened. To get the information "*how many cars are left in a closed parking*", we can connect the door sensor to the counter, and count "*door is opening*" and "*door is closing*" messages. If we reach two "*door is closing*" messages, then we can send an event containing the number of cars left. But it is difficult to know which one is closed (no sender identification). Either, BeC^3 doesnt provide any command to query a node, or to resend an information. BeC^3 is designed to build an automatic chain reaction depending on events. It does not provide dynamic requests facilities.

In fact, BeC^3 is not adapted to data-centric approaches. It is all about messages and events. BeC^3 concerns are about semantic (something is *full*, *hot*, *empty*, *new*, or a threshold is reached) and not about real values (17, 43...). However, BeC^3 is not exclusive. Other applications can run in parallel, for example DPWS, REST or other solutions, even proprietary.

6 BeC^3 implementation

In order to illustrate the innovative concepts of BeC^3, we designed the whole solution [4] that allows users to create, use and execute IoT applications using simple mashups and sharing tools. We also provide binaries for some devices to make them part of the BeC^3 application. The communication network here is based on XMPP, an experimental web implementation of D-LITe called D-LITe Cloud (offering virtual devices) and BeC^3 real supported devices.. This section presents how the software is used by the different kind of users.

6.1 BeC^3 tools running on nodes

The abstraction needed to offer a universal platform is provided by D-Lite. In our Crowd Centric organisation, this platform is made by *experts*, which represent 1% of the users (Fig 6). Each type of hardware runs its own version of D-LITe. We give some example code in Contiki-OS to help experts to create ports of D-LITe on new devices. We also provide binaries on our site : one version for TelosB[5], based on Contiki[6], and one for Android. D-LITe for TelosB gives access to it LED, button and temp sensor. It also provides optional computing capabilities (this logical part of BeC^3 managing variables depends on hardware processing capacities and is not always feasible). D-LITe for Android (Fig 12) allows the use of a widget corresponding to a button and notifications (vibration, pseudo LED) and to computing capacities. Eventually, we will include access to dialling, messages, camera, gps, etc.. We also propose a virtual node written in Java. It has all the features of a D-LITe node, and can be use as a real node (for example making a website reacts as an object, or accessing to social networks, or simply to offering a control through his computer). This program can easily be adapted and extended to offer new features, for example to create new widget or new services interacting with IoT applications build with BeC^3.

[4] BeC^3 WebSite - http://bec3.univ-mlv.fr/

[5] TelosB, a wireless sensor network device for experimentation and research http://www.memsic.com/

[6] an Open Source operating system for the Internet Of Things http://www.contiki-os.org/

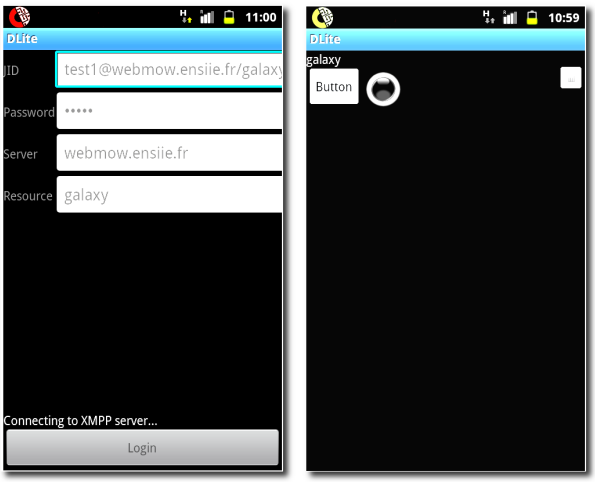


Fig. 12 D-LITe for Android. After authentication screen (on the left) that helps the identification of the device, the smartphone receives the logic it has to execute. In this example, this node has been remotely configure to offer a button and a pseudo-led that can be switched on or off by other nodes, according to BeC^3 uploaded logic (right screen).

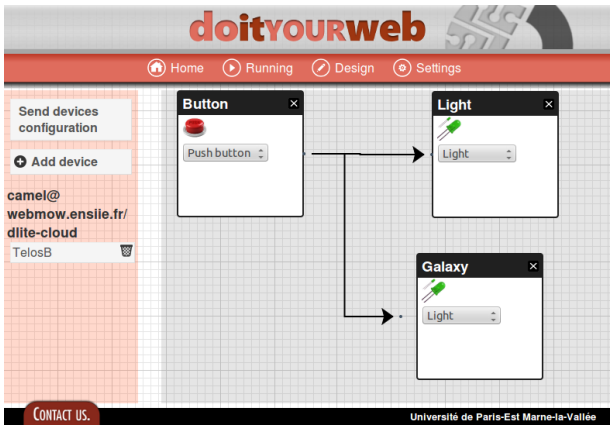


Fig. 13 The Design screen of BeC^3 mashup tool, showing all devices. The *behaviour* chosen for each of them is indicated on the option list field. Publish-subscribe relations are shown with arrows. One node (TelosB) is available (on the left). Application is deployed with "*Send devices configuration*".

6.2 Connecting nodes and composing application

Each node involved in an IoT application made with BeC^3 must be able to communicate with the others. For that purpose, we have chosen XMPP[7], a standardized protocol for real time communication. This protocol has caught our attention because it offers instant messaging and presence management. Thus, the discovery of new nodes is dynamic and their integration in the global structure is easy. To participate to an BeC^3 application, a node must be configured to connect to an XMPP server, by giving the account of the owner, his password, and the node's name (to be recognized as this specific node, i.e. *galaxy* in Fig 12). A user has access to all its nodes. If he becomes "friend" with another user, the friend's nodes can then be involved in any application. To create his application, we propose the BeC^3 mashup tool (Fig 13). When a user authenticates on this tool, he can list all the devices that are available on his account. It is possible to check and import others virtual nodes (such as proxy for accessing web services, etc) or that belong to another user (by giving credentials). Nodes available appear on the left side of the Setting screen (Fig 13). To involve an node in an application, BeC^3 user drags icons from left side to the central panel, and choose a Behaviour from a proposed list of compatible ones (not accessible in the first version of the program). Then, it is possible to link nodes to others, just by drawing arrows between them. Once it is finished, the user tries to send his choice to the nodes. After checking the consistency of the assembly (not available yet), the BeC^3 mashup tool send messages to each node in order to describe the logic it has to follow (the Behaviour) and the observer's list of that node (arrows).

[7] http://xmpp.org/

6.3 Putting all together

To test our implementation with some TelosB, some Android Smartphone, and a computer, one should use the following procedure:

- Download D-LITe binary for TelosB. Write user account, password, and name in the configuration file, and flash the Node. Only one flash is needed, because the logic is transmitted Over The Air.
- Install D-LITe application on Android nodes. This application asks for user credentials and node's name.
- run BeC^3 mashup tool) on the Computer. This tool asks for credentials (but only account and password). All nodes using this account appears in the setting screen of the application.
- The user compose its application using the nodes, choosing each Behaviour, and making links.
- When finished, his description is send to all nodes, and the application starts

This platform has enabled us to quickly write small applications involving TelosB, smartphones and virtual nodes. Besides the variety of tools used in the same application, BeC^3 has completely reconfigure nodes to dynamically build new applications in which the roles of each element could be very different. Hardware abstraction allows to dynamically combine wide range of materials and wide variety of uses. Obviously, this platform

is for demonstration use only. Further enhancements will follow as the BeC^3 community grows.

7 Conclusion

This paper presents BeC^3, our proposition to simplify the creation of Web of Things applications. In line with our previous work on the normalization of IoT application creation and deployment in smart networks in general and WSAN in particular, BeC^3 allows the identification and classification of many possible interactions between different behaviours present on each devices (or service). The abstraction of these exchanges offers the possibility to interconnect pre-written pieces of the application as long as they indicate the type of inputs and outputs that they manage. Finally, the BeC^3 sharing platform of pre-written behaviours simplifies the application creation process to an elementary and intuitive combination of compatible and semantically self-expressive bricks for heterogeneous types of hardware and software.

Hence, by reversing the well-known SOC paradigm where architects design new applications by combining existing web-services, we take advantage from the flexibility of our framework to deploy retroactively the necessary services needed for the application's execution. By providing a Crowd-Centric contributive system, we offer a very wide range of modular, scalable and incremental bricks of logic that could be endlessly combined to produce applications that could eventually build an open, collaborative and extensive Web of Things.

References

1. Fi-ware internet of things(iot) service enablement, 2011.
2. L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010.
3. F. J. Ballesteros, E. Soriano, and G. Guardiola. Octopus: An upperware based system for building personal pervasive environments. *Journal of Systems and Software*, 85(7):1637–1649, 2012.
4. D. Berardi, F. Cheikh, G. De Giacomo, F. Patrizi, and O. Ibarra. Automatic service composition via simulation. *International Journal of Foundations of Computer Science*, 19(2):429–451, 2008.
5. D. Brabham. Crowdsourcing as a model for problem solving an introduction and cases. *Convergence: The International Journal of Research into New Media Technologies*, 14(1):75–90, 2008.
6. T. Bultan, J. Su, and X. Fu. Analyzing conversations of web services. *Internet Computing, IEEE*, 10(1):18–25, 2006.
7. D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, and F. Patrizi. Automatic service composition and synthesis: the roman model. *IEEE Data Eng. Bull*, 31(3):18–22, 2008.
8. S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel. D-lite : Distributed logic for internet of things services. In *2011 IEEE International Conferences on Internet of Things, and Cyber, Physical and Social Computing*, pages 16–24. IEEE, 2011.
9. S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel. Services Collaboration in Wireless Sensor and Actuator Networks: Orchestration versus Choreography. In *17th IEEE Symposium on Computers and Communications (ISCC'12)*, page 8 pp, Cappadocia, Turquie, July 2012.
10. F. Curbera, Y. Goland, J. Klein, F. Leymann, S. Weerawarana, et al. Business process execution language for web services, version 1.1. 2003.
11. 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.
12. D. Green, R. Hulen, and J. Moody. IPV6 sensor service oriented architecture. In *Military Communications Conference*, pages 1–6, 2008.
13. D. Guinard and V. Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*. Citeseer, 2009.
14. Guinard, D. and Trifa, V. and Wilde, E. A resource oriented architecture for the web of things. *Proceedings of IoT*, 2010.
15. R. Gummadi, O. Gnawali, and R. Govindan. Macroprogramming wireless sensor networks using kairos. In *Distributed Computing in Sensor Systems*, pages 126–140. Springer, 2005.
16. J. M. Hernández-Muñoz, J. B. Vercher, L. Muñoz, J. A. Galache, M. Presser, L. A. H. Gómez, and J. Pettersson. Smart cities at the forefront of the future internet. In *The future internet*, pages 447–462. Springer, 2011.
17. N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web services choreography description language version 1.0. *W3C Working Draft*, 17:10–20041217, 2004.
18. A. Nayak and I. Stojmenović. *"Wireless sensor and actuator networks: algorithms and protocols for scalable coordination and data communication"*. Wiley-Interscience, 2009.
19. J. Nielsen. Participation inequality: lurkers vs. contributors in internet communities. *Jakob Nielsen's Alertbox*, 2006.
20. A. Pintus, D. Carboni, A. Piras, and A. Giordano. Connecting smart things through web services orchestrations. *Current Trends in Web Engineering*, pages 431–441, 2010.
21. M. Presser, P. M. Barnaghi, M. Eurich, and C. Villalonga. The sensei project: integrating the physical world with the digital world of the network of the future. *Communications Magazine, IEEE*, 47(4):1–4, 2009.
22. Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th international conference on World Wide Web*, pages 973–982. ACM, 2007.
23. Z. Shelby and C. Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley, 2010.
24. 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.

25. J. Su, T. Bultan, X. Fu, and X. Zhao. Towards a theory of web service choreographies. *Web Services and Formal Methods*, pages 1–16, 2008.
26. M. Welsh and G. Mainland. Programming sensor networks using abstract regions. NSDI, 2004.
27. S. Ziegler, C. Crettaz, L. Ladid, S. Krco, B. Pokric, A. F. Skarmeta, A. Jara, W. Kastner, and M. Jung. Iot6– moving to an ipv6-based future iot. In *The Future Internet*, pages 161–172. Springer, 2013.