Contents lists available at ScienceDirect
Pervasive and Mobile Computing
journal homepage: www.elsevier.com/locate/pmc

A resource oriented integration architecture for the Internet of Things: A business process perspective

Kashif Dar[a,*], Amir Taherkordi[a,b], Harun Baraki[c], Frank Eliassen[a], Kurt Geihs[c]

[a] *Informatics Department, University of Oslo, Norway*
[b] *R&D Department, Sonitor Technologies, Oslo, Norway*
[c] *Electrical Engineering and Computer Science Department, University of Kassel, Germany*

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Internet of Things
Business processes
Integration
Resource oriented architecture
Service composition
Events management
Mobile computing

ABSTRACT

The vision of the Internet of Things (IoT) foresees a future Internet incorporating smart physical objects that offer hosted functionality as IoT services. These services when integrated with the traditional enterprise level services form the creation of ambient intelligence for a wide range of applications. To facilitate seamless access and service life cycle management of large, distributed and heterogeneous IoT resources, *service oriented computing* and *resource oriented* approaches have been widely used as promising technologies. However, a reference architecture integrating IoT services into either of these two technologies is still an open research challenge. In this article, we adopt the resource oriented approach to provide an end-to-end integration architecture of front-end IoT devices with the back-end business process applications. The proposed architecture promises a programmer friendly access to IoT services, an event management mechanism to propagate context information of IoT devices, a service replacement facility upon service failure, and a decentralized execution of the IoT aware business processes.

1. Introduction

The vision of Internet of Things (IoT) targets the interconnection of physical objects in everyday life with the Internet in an effective, practical, and standard way [1]. With the emergence of Internet Protocol (IP) based IoT devices [2,3] and the concept of embedded Web Services (WS) [4,5], enterprise level applications such as *business processes* (BPs) are also being extended [6] to optimize their execution by collecting real time information provided by IoT devices. For instance, logistics efficiency [7], safety processes for storing hazardous materials [8], remote patient monitoring [9], and smart buildings [10] are a few application scenarios. Thus the goal of service-based IoT systems integration is to expose the real world IoT resources [11,12] in such a way so that they can be easily integrated into the virtual world of BPs. Such an integration opens up for a decentralized execution of IoT-aware BPs [10] due to large scale and massively distributed nature of these applications [11] and thus demands a programming-in-the-large based approaches [13]. On the other hand, this highlights a need for a uniform *architecture* facilitating an easy yet standardized access to IoT devices.

Service-based IoT systems are normally realized according to the principles of the Service Oriented Architecture (SOA) [11,10,14,7,5] or to the Resource Oriented Architecture (ROA) [15–17]. The SOA-based techniques provide a uniform and structured communication with low power IoT devices as in e.g., TinySOA [14], compression of the Simple Object

* Corresponding author.
E-mail address: kashifd@ifi.uio.no (K. Dar).

http://dx.doi.org/10.1016/j.pmcj.2014.11.005
1574-1192/© 2014 Elsevier B.V. All rights reserved.

Access Protocol (SOAP) messages for sensor devices [13,5], proprietary communication protocols [13], implementation of WS eventing [5], and compiling and deploying the Business Process Execution Language (BPEL) code up to the edge of the network [18,10,13,19]. All the aforementioned efforts provide extension to the WS-* family of standards as part of their approach. The ROA-based approaches exploit the Representational State Transfer (REST) design philosophy [15] to create, for instance, the Web of Things (WoTs) or simple mash-up applications [20,16,17]. The existing Web is already a good example of how REST endpoints interact with each other to build Web based applications.

Up to date, the above mentioned studies still largely lack a unified design for end-to-end integration of both enterprise level BPs and tiny IoT devices. This is especially true for ROA based approaches. The SOCRADES project [7] addresses this problem using the SOA-based approach. However, from the REST based approaches, we do not see any work addressing the problem of how to expose REST-based resources hosted by tiny IoT devices as composable services in enterprise level BPs. We believe that REST principles when applied to the integrated environment of IoT will form the foundation of smart environments in which sensing, computing, and communication can easily be mashed-up. This approach is being increasingly adopted by [16,21,22]. For instance, the Smart Energy Profile 2.0[1] is a REST-based standard used by devices from different vendors to seamlessly interoperate with the objective to monitor, control, inform and automate the delivery of energy and water.

In this article, we adopt the ROA approach for the design and implementation of a generic architectural model which provides core components required for an end-to-end integration of IoT systems with special focus on IoT-aware BPs. We present a standard-compliant approach to integrate resource constrained devices into Business Process Modelling Notations (BPMN)[2] based processes. The proposed architecture is characterized by promising: (i) necessary Application Programming Interfaces (APIs) to invoke IoT services which ease the life of IoT-aware BP developers, (ii) an event-based integration model built upon a well known publish–subscribe mechanism, (iii) a dynamic service replacement facility upon failure of IoT devices, and (iv) decentralized execution of the BP. In summary, our framework is a step forward to make tiny IoT services first class citizens so that they can be discovered like traditional WSs and can be easily accessed within the BP development environment.

In the remaining part of this paper, we first introduce an application scenario from the eHealth domain which highlights our main design requirements. In Section 3, we identify the relevant research challenges implied by these requirements. The proposed architecture is then presented in Section 4 along with the description of each involved component. In Section 5, we provide the implementation details of the proposed framework while Section 6 is dedicated to present some evaluation results. Finally, we discuss the related work in Section 7 while Section 8 offers conclusions as well as outlines our intentions of future work.

2. Application scenario

In this section we consider an Ambient Assisted Living (AAL) system supporting persons with a chronic illness or those with a need for constant medical surveillance (for instance, elderly people) in a way allowing them to continue to live independently at home. For this purpose, smart bio-medical devices are convenient to measure essential vital signs of the patient, e.g., temperature, blood pressure, heartbeat rate, cholesterol/O_2 level in the blood, and real time electrocardiogram (ECG) reading etc. As shown in Fig. 1, the whole BP is a composition of services (local or remote tasks) put together to provide the logic needed for the application. The BP is split into three parts: (i) the Set Top Box (STB), (ii) remote computer, and (iii) smart phone. The role of the STB is to locally compose and subsequently collect data from IoT services [10]. A part of the BP (installed on a stationary STB) transmits the measurement results collected from local IoT devices to a remote BP whenever it is actually needed. The remote BP not only provides an instant analysis but also long term remote patient monitoring. Patient data is then accessible by the doctor during a regular check-up; a diagnosis can be derived to provide remotely recommendations based on the day's health condition. In case of an emergency attention (e.g., blood pressure/sugar level has been dangerously high/low), real-time communication channels can be established to notify on-site health care providers and/or designated relatives.

Within the scope of the AAL system, we consider two main use cases: the indoor and outdoor patient environment. As an example of the former, suppose an elderly person suffering from diabetes wants to keep her doctor up-to-date about the sugar level in her blood. Upon measuring the current sugar level the smart bio-medical device via the STB transmits its reading to a remote hospital where her physician can access this information. Furthermore, if an abnormal data (violating the predefined sugar level threshold) is reported from her wearable units, the AAL system detects and alerts the physician at the hospital/clinic about the abnormal health condition of the patient. The remote physician can then trigger the analysis of sugar, insulin and O_2 saturation in her blood and send necessary medical prescription and other dietary recommendations to her mobile phone after checking this data.

The outdoor scenario takes place when the patient is outside her home. The services offered by implanted devices will no longer be available due to lack of communication link to the STB. In order to keep the system up and running the functionality offered by the STB that is needed to collect the required vital signs, should be delegated to the patient's smart

[1] http://www.zigbee.org/Standards/ZigBeeSmartEnergy/SmartEnergyProfile2.aspx.
[2] http://www.omg.org/spec/BPMN/2.0/.

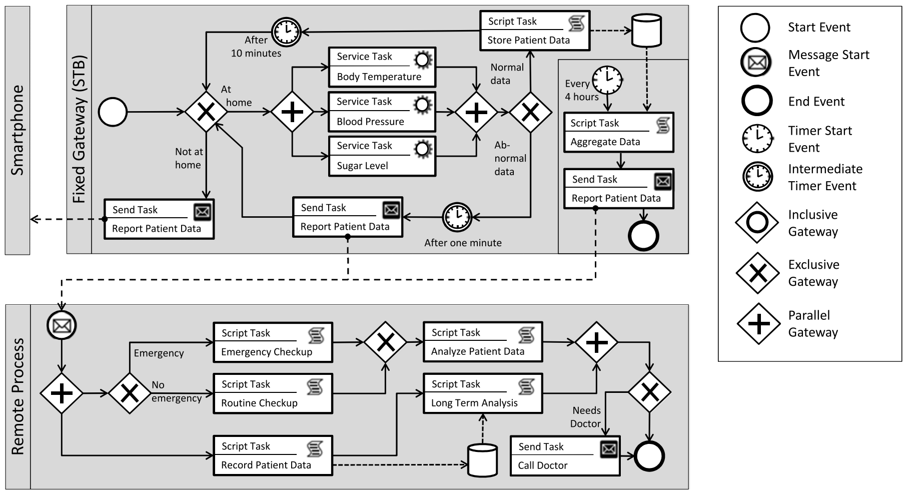

Fig. 1. A sample BP within the ambient assisted living system.

phone, assuming that she is carrying it while outside the home. Once this delegation process is completed, her smart phone will provide the necessary functionality to collect and process the required data from the attached devices. Similarly upon returning back home, the STB gets control back from the smart phone.

The application scenario presented above poses several technical challenges to disseminate real time information from the patient environment to remote hospital or other health care giver sites. First, the design and subsequent deployment of enterprise applications capable to consume services of smart IoT devices is a challenge. For example, if a BP developer wants to add a smart temperature reporting device into its application, how can he discover and embed its functionality as easily as when using the traditional WSs? Second, IoT applications are event driven by nature [12,5], therefore, an IoT aware BP must be reactive in the presence of different events generated in the IoT environment, e.g., an emergency situation in which the patient health is critically degraded must be reported to the remote physician. Third, such devices, due to their tender hardware characteristics and harsh operating environment [12], are more susceptible to failures. Therefore, a failure resistant mechanism is required to ensure continuity of the BP e.g., replacing a failed service with a similar one which is next available. Lastly, support of decentralized execution of the BP is crucial as we can see from the scenario of the patient being outside her home. In the following, we briefly describe the required design principles to meet these challenges.

3. Design principles and requirements

Defining automated processes to achieve business goals is a task that required until recently deep technical knowledge. The introduction of BP languages like WS-BPEL (Web Services Business Process Execution Language)[3] and BPMN enabled domain experts to easily implement processes without knowing much technical details. However, current BP software suites are not intended to support domain experts in developing IoT-aware processes. Our goal is to extend a BP software suite in such a way that the fundamental features of IoT systems are leveraged without modifying the BP language and its characteristics. The following design principles help us to define the main requirements in this context.

Unified integration. The real world services offered by IoT devices should be accessible via the Web to integrate into the enterprise level BPs. This means that the proposed architecture must support IoT devices as WSs that can be treated as potential resources accessible through unique URIs. Further, in order to expose such services at Internet scale, these should be described through a standard service definition language to exhibit their functional as well as non-functional aspects. To achieve uniformity and interoperability between different services platforms, they must be accessible through standard communication protocols, e.g., 6lowPAN (IPv6 over low power Wireless Personal Area Networks) [3], HTTP and/or Constrained Application Protocol (CoAP[4]). The latter is a RESTful protocol [15] which is used as a replacement of HTTP for low power devices [23]. Lastly, to increase the programmability of IoT devices, the necessary service artefacts should be *automatically* generated. The services artefacts are pieces of executable code or document files that facilitate an easy integration of services provided by IoT devices into the BP with a minimum programming effort and without knowing the lower level implementation details about underlying communication protocols.

Support for event-based interaction. Due to event driven nature of IoT applications [12,5,11], real world services deployed on IoT devices must support event based asynchronous interaction with BPs. Considering resource efficiency, the proposed

3 http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
4 https://tools.ietf.org/html/rfc7252.

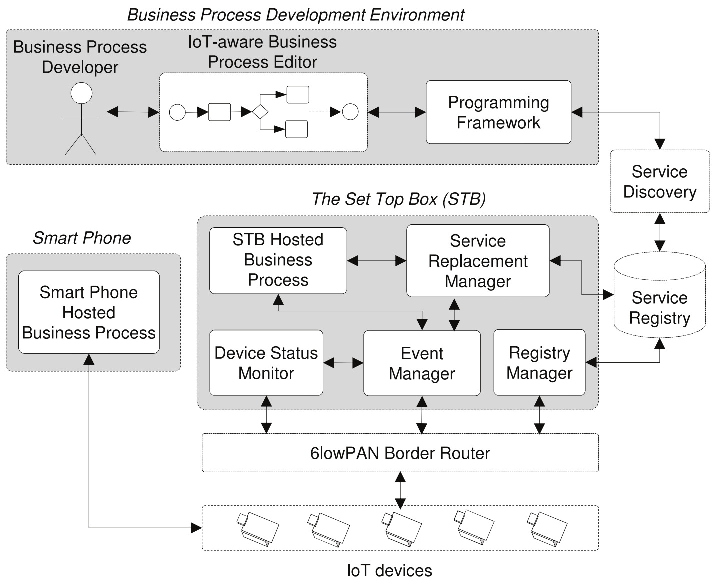
Business Process Development Environment
Business Process Developer
IoT-aware Business Process Editor
Programming Framework
Service Discovery
The Set Top Box (STB)
Smart Phone
Smart Phone Hosted Business Process
STB Hosted Business Process
Service Replacement Manager
Service Registry
Device Status Monitor
Event Manager
Registry Manager
6lowPAN Border Router
IoT devices

Fig. 2. The proposed ROA framework.

event mechanism should be supported by a *device-push* interaction, i.e., the BP should not have to poll for detecting device-level events. Such an interaction would help to extend the battery life time of the device since the device only needs to report an event when it actually happens. Furthermore, only relevant data that is of real interest to the BP should be reported. This can be achieved through intermediate data processing and filtering. For example, to report a high blood pressure, the BP can be alerted only when the blood pressure device reports a value that exceeds a predefined threshold.

Volatility of IoT services and service replacement. Within the IoT environment, the services provided by IoT devices have more volatile availability as compared to traditional WSs [12]. For example, low device battery level or broken communication links can easily lead to service unavailability. Once a failure of a participating IoT service is detected, the failed service should immediately be replaced by the next available service (if any) that offer similar functionality. However, to ensure a smooth replacement process, the current execution state of the BP must be retained so that it can resume its execution after the service replacement.

Decentralized business process. As discussed in the outdoor scenario in Section 2, the centralized architecture of IoT-aware BPs poses difficulties in adapting to changing operating environment of smart devices [10,7]. Thus, our requirement is not only to deal with a centralized (orchestration) but also a decentralized (choreography) execution of the IoT aware BP. The former assumes a single BP controlling the execution of participating services while latter enables the domain expert to design and simultaneously define the communication between different BPs.

4. Proposed ROA architecture

To realize the above design principles, we explore the design space relevant to manage the life cycle of standard WSs embedded into IoT devices. The proposed framework not only features access to physical entities but is also designed around the BP developers' perspective to facilitate an easy integration using simple APIs. Fig. 2 depicts the main components of our ROA framework. The services exposed by IoT devices (at the bottom) are integrated with IoT-aware BPs (at the top) through Programming Framework and several other components deployed on the STB (in the middle). In the following subsections, we will describe these components in detail.

4.1. Service description

As shown in Fig. 3(b), the service is a set of resources that expose their functionality via interfaces. Service descriptions within our framework are written in a standard language such as Web Services Description Language (WSDL) 2.0[5] or Web Application Description Language (WADL[6]). A service description consists of a human-understandable name and a base URI containing host name and port number of the network resource in which the service endpoints are contained. Each resource is described by the following attributes: resource name, interface type e.g., REST or SOAP, path along with the service base

5 http://www.w3.org/TR/wsdl20-primer/.
6 http://www.w3.org/Submission/wadl/.

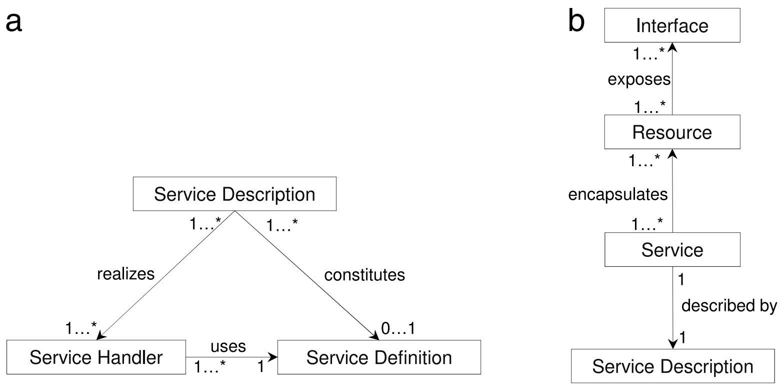

Fig. 3. The service description (b) and the service artefacts (a) model.

URI provides a full URI to access the resource, and link(s) to its exposed interface(s). Each interface further describes which methods are supported by a resource and which input/output parameters are needed to invoke these methods. The intention of such a design choice is to bring flexibility by allowing multiple resources to share a set of uniform interfaces and then generate a final service description by integrating the descriptions of both resources and their interfaces. Ultimately, the service description must be stored into the service registry where it is accessible through a discovery protocol whose design is out of the scope of this work.

4.2. Programming framework

To fulfil the goal of seamless integration of IoT devices within the BP development environment, our framework targets to provide: (i) necessary APIs, and (ii) services artefacts to access the IoT provided services just like a method call of an object. By using our framework, the application developer achieves this goal in two logical steps. In the first step, to locate IoT services on the Web, the application developer specifies a search criterion to Programming Framework. Programming Framework, with the help of an external *service discovery* mechanism, fetches a service description file for each available candidate service (assuming that there may be more than one service available to fulfil the given functionality). In the second step, the application developer selects only those services s/he needs to fulfil the desired application scenario. Programming Framework then processes the services description files of the selected services and automatically generates a *service handler* file for each participating IoT service and a single *service definition* file (as shown in Fig. 3(a)). The service handler file provides a simple API to access its corresponding IoT service, while the service definition file contains the meta-data of all the included services. This metadata information can be further used to provide several features in the BP development environment, e.g., *quality of service* information about each service. We describe the service artefacts in more detail in Section 5

4.3. Event Manager

The purpose of Event Manager is to act as the communication substrate of the framework supporting event-based interaction among the entities of the framework. This kind of interaction is commonly referred to as publish–subscribe systems. A publish–subscribe system is a system where publishers publish structured messages (events) to an event service (Event Manager) and subscribers express interest in different events to the event service through subscriptions which can be arbitrary patterns over the structured messages. For example, a BP could subscribe to events from a particular IoT service (e.g., the temperature service), while Service Replacement Manager (described in Section 4.5) would be interested in receiving events indicating IoT devices failure. IoT services would typically act as publishers publishing events about their status (e.g., blood pressure or temperature), while Device Status Monitor (described in Section 4.4) would publish events about devices failure. A main property of publish–subscribe systems that we are interested in, is that they are space-decoupled meaning that the publisher does not need to know the identity of the subscribers. This allows for a highly flexible and dynamic architecture. Publish–subscribe systems are readily available off-the-shelf and can be used to realize Event Manager of our framework.

4.4. Device status monitor

To support a failure-transparent execution of IoT aware BPs, our framework is equipped with timely reporting of service failures within the IoT environment. To fulfil this, Event Manager mediates between Device Status Monitor, the BP (assumed only single process at this stage), and Service Replacement Manager[7] (described next). Device Status Monitor then registers

[7] The BP initializes Service Replacement Manager by informing about participating IoT devices.

for publishing notice about possible failure of each participating IoT device while the BP and Service Replacement Manager both subscribe for receiving such publications.

After this phase, each participating device sends a periodic heartbeat message to Device Status Monitor showing its liveness. Whenever a device fails to send heartbeat message after a given time-out, Device Status Monitor reports its unavailability by sending *device id* to Event Manager. Event Manager, upon receiving such a message, notifies not only the concerned BP but also Service Replacement Manager in order to replace the unavailable service.

4.5. Service Replacement Manager

The goal of Service Replacement Manager is to first discover substitute services that may replace the unavailable service. To achieve this, the replacement manager requests the service registry for a list of candidate services. Following, the service registry looks for services whose description matches with that of the unavailable service. Out of the set of candidate services that resulted from the previous phase, the replacement manager then randomly selects one service that can actually substitute the unavailable service. The important task during service replacement is to ensure the state synchronization of the BP so that it resumes its execution from the same state as of before its activities were blocked due to service failure. To fulfil this, a periodic check pointing is made in order to retain the BP state and when the service replacement is completed, the execution of the BP is resumed from the latest checkpoint state.

4.6. Distributed business processes

To enable the application of BPs in our framework, we need an appropriate language and engine to define and execute them. We will discuss our decision in this subsection and explain subsequently how the BP engine will be employed.

The processes in our framework are designed by means of BPMN 2.0. In a pure workflow orchestration language, such as WS-BPEL, a domain expert would create one central process that invokes different services. This procedure does not meet our requirements, in which several processes running on different devices shall interact with each other. We decided to use BPMN since it supports orchestration as well as choreography. The latter allows the domain expert to define the communication between the BPs as well. In comparison to WS-BPEL, BPMN also supports a larger set of workflow patterns and events, has a standardized diagram interchange format, and provides user tasks for human interaction [24]. Furthermore, BPMN can be extended by specifying new domain-specific elements so that future process developments may be simplified. Last but not least, BPMN 2.0 was accepted on July 2013 as international standard (ISO/IEC 19510:2013).

A further option would be to apply rule-based languages like Simple Rule Markup Language (SRML) [25], Semantic Web Rule Language (SWRL) [26], or the Drools rule language [27]. However, they do not share the extensive capabilities of BPMN to describe a workflow and the interaction with other peers and services. For this reason, rule-based languages are often combined with BPMN and WS-BPEL [28]. In our case, rules could describe the business logic while BPMN would take care of the program logic. This would foster a separation of concerns with the advantage that modifications of the business logic would not require accessing the BPMN code. Therefore, BPMN provides the *Business Rule Task* activity to integrate rules and rule engines. The BPMN engine we are using supports Drools, but unfortunately Drools is not runnable on current versions of Android. Hence, we describe the business logic currently within the BPMN file by using BPMN activities and the Java Unified Expression Language Juel,[8] which is a simple scripting language. Alternatively, a rule engine like JESS[9] for Android can be used. In the current phase, the memory consumption of such engines is too high since most of them use the widespread Rete algorithm [29], which compares a set of patterns with a set of objects in the working memory.

To take advantage of the choreography abilities of BPMN for our framework, we need a BPMN engine not only on the STB and the remote BP server, but also on the smart phone of the patient. This allows the domain expert to create an adapted BP for the smart phone that is communicating with the sensors attached to the patient even if the patient is outdoor. As shown in Fig. 1, the domain expert has also to adjust the BP running on the STB in such a way that the monitoring of the patient is delegated to the smart phone in case the patient is leaving home. For example, if the patient's smart phone is no longer accessible via the home network, it can be assumed that the patient is out of reach for the STB. That means that the domain expert needs to incorporate an activity in the STB's BP that is checking for the smart phone's local availability. If it is not on home site, the BP on the STB may send a message to the smart phone. The message can also include variables that should be initialized on the smart phone's BP. As soon as the message is received by the smart phone, the BP is started. Fig. 4 depicts an example for a BP on a smart phone.

5. Implementation

The implementation of the proposed architecture requires dealing with several programming tasks from design of the desktop applications to the lower level IoT devices programming. First, to ensure rapid application development, necessary

8 http://juel.sourceforge.net/.
9 http://www.jessrules.com/jess/index.shtml.

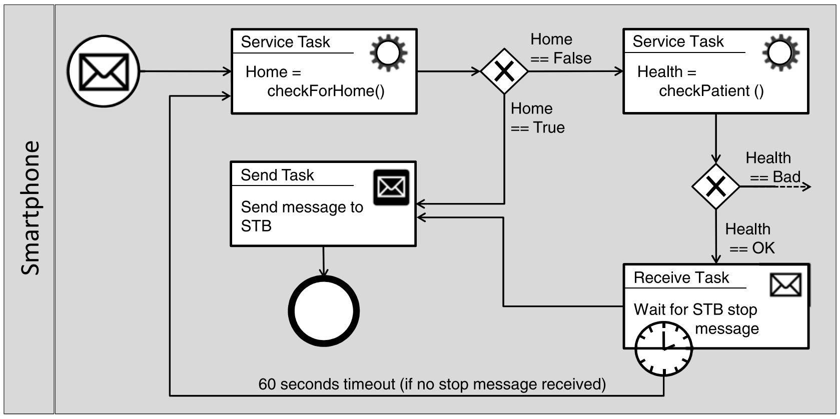

Fig. 4. The BP running on a smart phone.

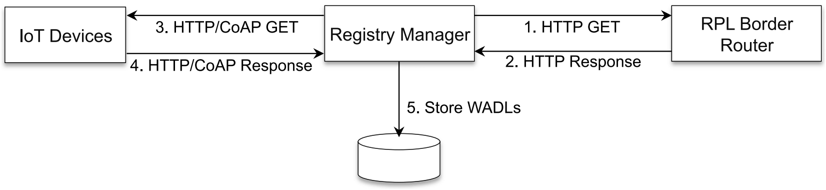

Fig. 5. Resource discovery by Registry Manager.

programming abstractions are provided at application level. Second, the STB[10] components (shown in Fig. 1) are developed to locally collect and process data originated from IoT devices. Third, at the device level, we deal with programming low-powered sensor devices.

5.1. Developing tiny RESTful WSs

We developed WSs on the Contiki-based[11] Tmote Sky[12] nodes. These services are developed following the characteristics of a service in our base platform. That is, each service provides access to resources through an RESTful API and implements at least one of the generic interfaces. As REST application protocol, we used both HTTP and CoAP. For the latest CoAP[13] (server side) implementation, we considered the Erbium[14] source code.

After the deployment of RESTful WSs, the next step was to expose them so that they can be discovered at the Internet scale. Registry Manager, developed in Java, enables a REST server (device hosting a set of REST endpoints) to be exposed as RESTful service. This is done in two steps: (i) resource discovery and then (ii) automatic generation of service description. The former is achieved through Contiki's built-in REST engine which implements the CoAP standard resource discovery mechanism: each REST server implements a default resource called *.well-known/core* (WC). A list of hosted REST resources is returned upon making a GET request to the WC resource.

As depicted in Fig. 5, RM enables resource discovery by first making an HTTP GET request to the WC resource of the 6lowPAN border router, which in response returns the list of attached devices. RM then sends a unicast GET request to WC resource hosted by each neighbour device to obtain the list of REST resources that the underlying REST server provides. The received response is further parsed by the RM to generate a service description in the form of a WADL file as shown in Fig. 6.

5.2. IoT-aware BP development

To demonstrate the feasibility of integrating smart objects into the BP, we used BPMN compliant jBPM (Java Business Process Management)[15] software suite as a service composition environment in which the application logic is expressed

10 Currently implemented on a Ubuntu computer.
11 http://www.contiki-os.org.
12 http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf.
13 http://tools.ietf.org/html/draft-ietf-core-coap-13.
14 C implementation of CoAP-13 draft: http://people.inf.ethz.ch/mkovatsc/erbium.php.
15 http://www.jboss.org/jbpm.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<application>
   <resources Id="12" base_uri="coap://[aaaa::212:7400:1360:c66b]:5683">
      <resource Ttile="Temperature sensor(supports JSON)"
       if="/home/kashifd/jbpm_workspace/interfaces/get_if1"
       path="/sensors/temperature" rt="TemperatureSensor">
          <methods>
             <method name="GET">
                <output_param name="temperature_reading" type="Degree"/>
             </method>
             <method name="PUT">
                <input_param name="temperature_threshold" type="Degree"/>
                <output_param name="temperature_reading" type="Degree"/>
             </method>
          </methods>
       </resource>
    <resources>
</application>
```

Fig. 6. A WADL file generated by Registry Manager.

```
/**
* Auto generated Temperature Sensor (resource) class that extends
Resource class
**/

public class TemperatureSensor extends Resource {
    //Declare private members here

    // through default constructor, set resource full path
    public TemperatureSensor() {
       super(coap://[aaaa::212:7400:1360:c66b]:5683, /temperature);
    }

    // instantiate CoapClient to make a CoAP request
    CoapClient coapClient = new CoapClient();

    public Degree performGet(Degree temperatureThreshold) {
      // Local vaiable(s)
       String coapResponse;
       Degree result;

       //TODO: Before making CoAP request, formulate the
      // CoAP request based on the input parameters

       //receive CoAP response
       coapResponse =
this.coapClient.sendCoAPRequest(super.getCompletePath(), GET)

       //TODO: Process coapResponse to get result

       return result;
    }
}
```

```
import
org.drools.process.core.datatype.impl.type.StringData
Type;

[
  // the Notification work item

  [
    "name" :"Temperature_GET",

    "parameters" : [
      "temperature_threshold": new DegreeDataType()
    ],
    "result" : [
      "temperature_reading": new DegreeDataType()
    ],
    "displayName" :"TemperatureSensor",
    "icon" : "icon path"
  ],
  [
    "name" :"LightSensor_PUT",

    "parameters" : [
      "light_color": new StringDataType()
    ],
    "displayName" :"LightSensor",
    "icon" : "icon path"
  ],
.
.
.
]
```

Fig. 7. The service handler (on left) and service definition (on right) files.

by composing local and remote service tasks using the BPMN based workflow model. Whenever, the application designer needs to include IoT services into the BP, s/he first searches the available services through an external service discovery protocol (e.g., jUDDI[16]) to fetch the WADL files for available services. Programming Framework component (developed in Java) then generates the required service artefacts, i.e., *service handlers* and *service definition* files (described in Section 4.2) by processing these WADLs. The sample code excerpts of these files are shown in Fig. 7. With the help of generated services artefacts, the jBPM integrated development environment then displays the discovered IoT services into the available service pallet. Finally, the application developer can easily add these services into the composition logic using simply drag and drop facility. In order to invoke these IoT services, the service handler (generated for each service) provides a simple API to invoke the REST endpoints of the associated service. This API is backed by the Java based Californium[17] CoAP (client side) implementation.

5.3. *Event manager*

To implement Event Manager, we used PADRES (Publish–Subscribe Applied to Distributed Resource Scheduling) system which is developed by the Middleware Systems Research Group[18] at the University of Toronto. Within our application scenario, the PADRES broker (PB), from one end, receives requests from IoT devices for the potential publications (called advertisements) while on the other hand, the BP can subscribe for a particular publication by specifying a subscription criterion. For example, the BP can subscribe for a high blood pressure event (blood pressure exceeds a particular threshold).

16 http://juddi.apache.org/.
17 http://people.inf.ethz.ch/mkovatsc/californium.php.
18 http://msrg.org/.

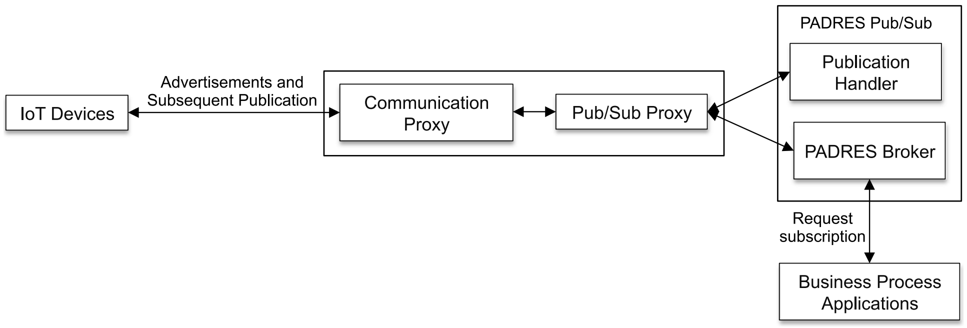

Fig. 8. Realization of event management model.

Fig. 8 depicts how the PB mediates between the physical world of smart IoT devices and the virtual world of BP applications. An IoT device, through the Californium CoAP client (the *communication proxy*), first sends an advertisement request to the Publish–Subscribe Proxy (PSP) (we developed) to interact with the PADRES system. This advertisement request contains the *device id*, the IPv6 address, of the device and a *dummy payload*. The PSP then, with the help of PADRES *publication handler* register this request by storing the *device id* in the PSP table. Similarly, a BP first sends an HTTP request to register its subscription with the PB. It is then the job of PB to filter and forward oncoming publications to intended subscribers. In this way, multiple BPs can subscribe for publications from multiple IoT devices and vice versa.

5.4. *Service replacement manager*

When the BP is notified by Event Manager (through a call back function) about the failure of a particular device. At this point, the BP is halted (by setting it into a wait state) until the failed service is replaced. In the meantime, Service Replacement Manager performs the following:

(1) *Select a candidate service:* Service Replacement Manager first fetches the candidate services from Service Registry and randomly selects one of them for substitution. In case of no alternate service is available, an exception is generated to report the domain expert responsible for BP maintenance. In addition, Service Replacement Manager performs a syntactic comparison between the BP composition logic and the failed service in order to determine its exact syntax used in the composition (e.g., the signature of a method call). This is done by parsing both the XML schema of the BP and the WADL file of the failed service.

(2) *Perform the actual service replacement:* We used two methods for performing the service replacement: (i) replacing the whole process with another instance containing newly configured services, and (ii) re-binding only the substitute service. The former is achieved by modifying the available XML schema (the process definition) of the whole process and instantiating it at runtime through a special feature of *process versioning* offered by jBPM. The latter is accomplished with the help of a special *proxy* service. The BP contains only abstract (without implementation) services and always invokes the concrete (with implementation) services through this proxy. In order to rebind the substitute service, Service Replacement Manager only configures the proxy service. We will further discuss the trade-off between these two approaches in the evaluation section.

5.5. *Distributed business process execution*

A key challenge during our work was to port a BP execution engine to Android. These engines have normally a great number of dependencies on some of the Java runtime libraries that were not available in Android 4.3.[19] Besides, the engine has to be economical concerning storage, memory and CPU load since smart phones are resource restricted. With these aims in mind, we analysed several execution engines and opted finally for the Activiti BPMN execution engine.[20]

Activiti is a lightweight BPMN engine that is able to run without an application server. However, it requires a database that supports JDBC (Java Database Connectivity)[21] to store various information like monitoring data, the states of paused BP instances, and the BP itself. Additionally, Activiti makes use of the Java XML stream reader and other libraries that were not available for Android. In consequence, we adapted the Activiti source code to Android's XML pull parser,[22] to Android's SQLite database,[23] which does not support JDBC (Java Database Connectivity),[24] and to numerous other libraries available

19 https://developer.android.com/about/versions/android-4.3.html.
20 http://activiti.org/.
21 http://www.oracle.com/technetwork/java/javase/jdbc/index.html.
22 http://developer.android.com/reference/org/xmlpull/v1/XmlPullParser.html.
23 http://developer.android.com/reference/android/database/sqlite/package-summary.html.
24 http://www.oracle.com/technetwork/java/javase/jdbc/index.html.

on Android. The resulting BPMN engine on Android does not support the full scope of features that the desktop version of Activiti is covering. For instance, we did not include the libraries for the Spring framework.[25] However, the BPMN process language is fully supported.

The next step was to establish a communication link between the BPMN engine on the smart phone and the engine running on the STB. We chose MQTT (Message Queuing Telemetry Transport) as it is a very lightweight but reliable publish–subscribe message protocol [30]. It is optimized for resource constrained devices and takes particularly unreliable networks, low bandwidth and limited battery into account. In June 2014 the first candidate OASIS Standard was published by the OASIS Technical Committee (Organization for the Advancement of Structured Information Standards).[26] The current version of MQTT (3.1.1) is agnostic to the content type of the payload. The OASIS Committee announced to introduce an enhanced message payload typing. Currently, we serialize objects in our framework to transfer them via MQTT.

Using the different features of Activiti, especially its *Runtime Service*, the current state of a process instance can be persisted and sent via MQTT to a receiver with a running Activiti engine. The sending party would stop its process instance while the receiver would continue the process at the same position with the same variables available at the sender. BP itself can be transferred through the *Repository Service* of Activiti. Nonetheless, we decided not to make use of these services because the migration of process instances implies the same process definition on the stationary computer and on the mobile device. However, a BP running on a smart phone may use different services, scripts and activities than those on a desktop computer, thereby they should be tailored to the current context and needs of the mobile user. In addition, the domain expert would not have any control over whether and when the process instance should migrate. Furthermore, the application of Activiti services would impede the usage of other BPMN engines, such as jBPM, on the other participating parties. Hence, we delegate this task to the domain expert. S/he should decide how the BP on the smart phone should look like and when there should be a switch between the participating processes. To be more precise, it is not about migration but delegation. As shown in Fig. 4, in our scenario the BP on the smart phone will transfer its data to the BP on the STB and stop as soon as the user arrives home. The other way round, the BP of the STB checks whether the conditions of a switch to the smart phone are met and, if so, the BP will serialize the required variables and transfer them via MQTT to the BP of the smart phone. Thus, the switch is executed by the BPs themselves and between different BPs on different peers instead of being executed by external services between same BPs on different peers.

The MQTT service running on Android would receive the message sent by the BP on the STB and would forward it to a *Message Start Event* or a *Receive Task* of the smart phone's BP. It should be noted that due to the lack of Zigbee-based communication in most of smart phones, our prototype currently uses Bluetooth to communicate with the sensor devices attached on the patient's body.

6. Evaluation

After the implementation of our framework, we performed evaluation to demonstrate its feasibility to be deployed in real world scenarios. To the best of our knowledge, most of the existing service-based integration frameworks for IoT [11,14,21] provide only the implementation of the presented case studies. There exists only few evaluations that provide insights about fine tuning the Medium Access Control (MAC) layer [2], optimization of the Transport Control Protocol (TCP) [5], memory footprint of an automatically generated code for integrating Wireless Sensor Networks (WSNs) [13], and the comparison between RESTful and SOAP based WSs [2].

By carefully inspecting the aforementioned evaluations, we considered both qualitative and quantitative aspects for the evaluation of our proposed framework. Regarding the qualitative aspect of our work, we made a step forward to expose the IoT services, thanks to our RM, to the standard WSs discovery protocols. At the same time, we exploited state-of-the-art technologies designed especially for the Future Internet, e.g., CoAP and 6lowPAN to bridge the connectivity gap between IoT systems and the enterprise applications. In addition, Programming Framework offers programming abstractions for application developers to rapidly develop IoT aware BPs with minimum knowledge of underlying glueing technologies.

Regarding the quantitative evaluation, we mainly investigated: (i) latency (end-to-end delay) caused by EM, (ii) comparison between two approaches adopted for service replacement, and (iii) processing overhead of the BP running on the android smart phone. Below, we first describe the test-bed design that we used to perform our experiments and then present the achieved results.

6.1. Testbed design

We divided our testbed into two main parts: (i) network of both real and simulated sensor devices and (ii) design of BP support on a smart phone. A short description of each follows:

Network of real and simulated sensor devices: We used Contiki based Tmote Sky nodes for both real and Cooja[27] simulation environment. The platform is based on an 8 MHz MSP430 microcontroller, a CC2420 radio chip, 48 KB of programmable flash

[25] http://spring.io/.
[26] https://www.oasis-open.org/.
[27] http://www.contiki-os.org/start.html#start-cooja.

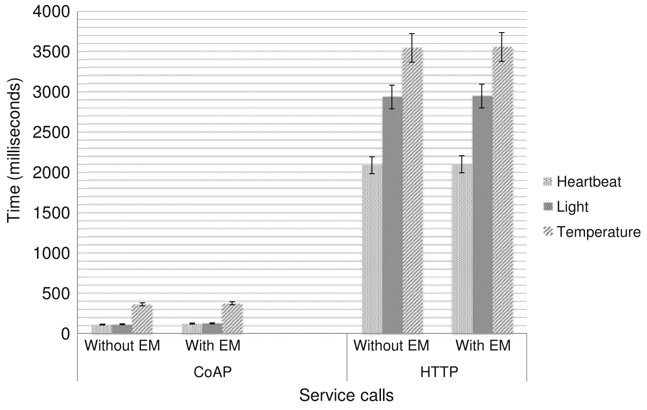

Fig. 9. The average end-to-end latency with and without Event Manager using single sensor device.

memory and 10 KB of RAM. We use a small single-hop network with static routes and a varying number of sensor nodes. Among these nodes, one node implements the 6LoWPAN border router connected via USB to a Ubuntu computer. Finally, we implemented three different types of services: (i) heartbeat: sending a dummy payload, (ii) light, and (iii) a temperature service on a single Tmote sky node.

BPMN execution engine on a smart phone: We installed the Activiti BPMN engine on a Samsung GT-I9300 with 1 GB of RAM and a quad-core processor with 1.4 GHz. We used Activiti 5.13 and ported it to Android 4.1 as explained in Section 5.5. BP depicted in Fig. 11 was deployed to the smart phone. The process does not invoke external WSs since we want to minimize external influences such as network latency and computing time of external devices. By performing this experiment, the resource consumption and execution time of the BPMN engine on Android is measured. The so called *script tasks* in our BP execute some assignments but do not interact with external services.

6.2. Latency evaluation

The purpose of evaluating the latency caused by Event Manager is to check the feasibility of a publish–subscribe based event notification mechanism within constrained IoT environments. We performed experiments using both real and simulated devices for different scenarios described below.

Experiments with real devices: We performed two types of experiments with real devices. For the first type, we considered that only one node has subscribed for the publication to the EM. Similarly, on the other hand, a single BP is a subscriber. We repeated this experiment for all the three types of services (mentioned above) on the device. In order to have a performance comparison within our framework, we used both CoAP and HTTP protocols. The results presented in Fig. 9 demonstrate the average latency over 10 readings. The HTTP suffers greater overhead due to increased message size, however, we see the overhead caused by Event Manager is only 12 ms which can be ignored compared to the total end-to-end delay.

For the second type of test, we used multiple sensor devices that publish only CoAP messages. The purpose of performing this type of test is to ensure that our solution is scalable when the number of sensor devices increases. We used seven sensor devices in total where each device publishes *temperature* data that has been subscribed by five different BPs. We ran the test until a few hundred publication messages are generated by each sensor. To measure an end-to-end latency for each sensor, we calculated the average over 50 readings i.e., 375 ms in which the average latency caused by Event Manager is only 15 ms which is only 3 ms increase from the first type of test.

Simulation experiments: Using Cooja, we performed further scalability tests by increasing the number of sensors up to 30 which we considered could be the maximum number of sensors within the application scenario that we envisioned in Section 2. From this test, we noticed that the latency due to Event Manager increases linearly with the increase in the number of sensors. However, this is still insignificant compared to the total end-to-end delay. Fig. 10 shows the average values of latency (Y-axis) with different number of sensors (X-axis).

6.3. Cost of service replacement

As explained in Section 5.4, the process of replacing the whole BP with newly configured one is relatively easy task as compared to fine-grained reconfiguration in proxy based approach. In this sub-section, we will compare the computation overhead caused by each approach. Note that we did not consider the lookup time to search for candidate services since it can be ignored for limited number of target devices. In the following, we quantify the computation overhead for each of these two approaches described in Section 5.4.

Replacing the whole process: We calculated the total time of process instantiation by using a simple jBPM process consuming only two services from sensor devices. The average (over 10 readings) instantiation time for the jBPM process was 1600

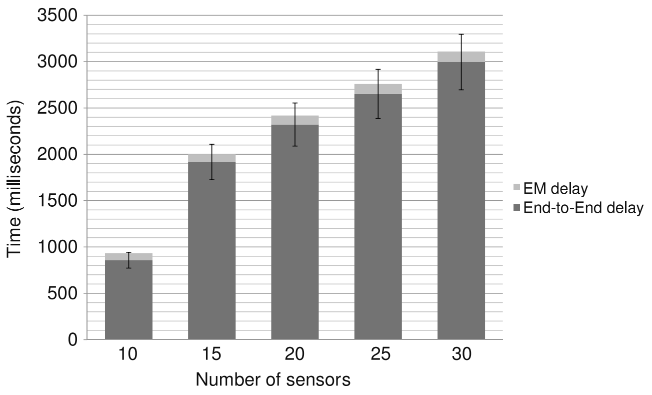

Fig. 10. Event Manager overhead for different number of Cooja nodes.

ms which includes time to (i) reconfigure the process XML schema, (ii) process state initialization, (iii) create the stateful knowledge session,[28] (iv) register the service handlers (described in Section 4.2), and (v) start the execution of the process.

Proxy based approach: Given the substitute service, the time taken to perform proxy based re-configuration includes creation of new service handler for the substitute service and replacing it with the failed one. We observed that the average (over 10 values) time to re-configure the proxy is 80.5 ms.

From the above results, we see the proxy based reconfiguration is more effective than the replacement of whole process, hence making it a potential candidate for IoT environments where frequent service failures are observed.

6.4. BPMN engine on Android

A patient who is using his smart phone to be monitored outdoor, is entitled to expect that the monitoring application does not reduce the battery capacity immoderately. In general, a main prerequisite for applications running on a smart phone is that they have to be energy efficient. This means that the application's CPU overhead, memory and storage consumption and its network activities have to be tailored to the limited energy source. At the moment, we have not applied any special optimizations for the BPMN engine. However, we are using MQTT and SQLite that are both suitable for mobile applications. The SQLite database is already included in Android and MQTT is a lightweight publish–subscribe message protocol designed for resource-constrained devices. Furthermore, certain libraries, such as the logging and the spring libraries, were removed and not used anymore.

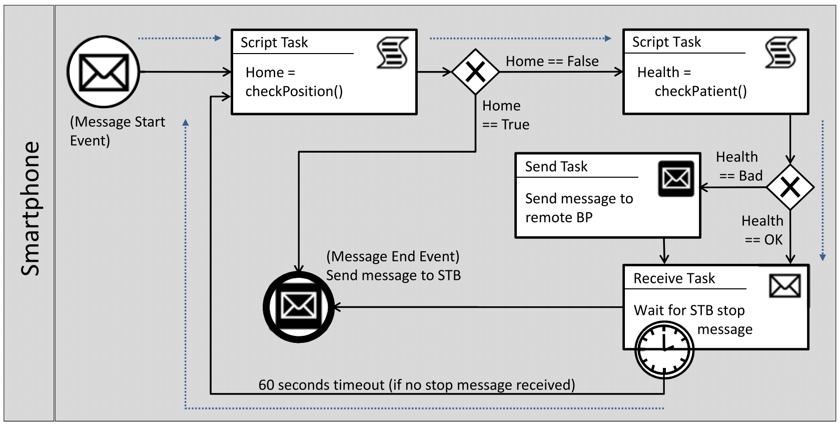

Fig. 11. The BP running on the smartphone.

Starting the BPMN engine that includes the set up of the database and the start of the MQTT service, requires 2.9 s on our smart phone. This means that after 2.9 s the first XML file is read in and executed. The XML file is converted to Java and stored in the database in 150 ms. Now, the process can be triggered by a MQTT message. After receiving a message, a process instance is created and executed. Our sample process (Fig. 11) needs 210 ms from the *message start event* to the

[28] Allows the application (BP) to establish conversation with the underlying execution engine.

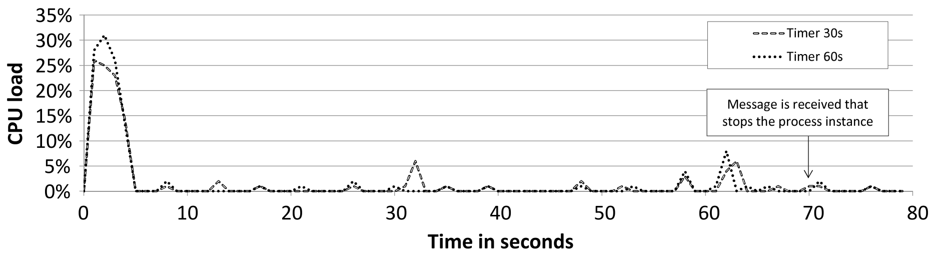

Fig. 12. The smart phone CPU load during running BP.

receive task. Part of this time overhead is due to the *receive task* which is a waiting state that initiates Activiti to persist the process instance and store it in the database. The waiting state of the *receive task* can be interrupted by an external message, e.g., from the STB at home. In our scenario, an interrupting *timer event* is also attached to the *receive task*. If no message is forwarded to the *receive task* by the MQTT service during a defined time limit, the timer will interrupt the *receive task* and continue the process. The dotted lines in Fig. 11 illustrate the path. Using the *timer event*, the time between the health checks can be configured. The process loops and stops again in the *receive task* if the patient is not at home. The execution of one loop needs 305 ms on average. In Fig. 12, the CPU load of the process is illustrated in case that the timer limit is set to 30 s in the first scenario and 60 s in the second scenario. The chart shows clearly that the CPU load coincides with the waiting states. During a waiting state the CPU load falls below 2%. In both scenarios, the BPMN engine was allocated 17.4 MB of RAM and 400–500 KB of cache for temporary files and database tables. The installation itself needs 7.55 MB of disk space. Given the fact that the current smart phones often have more than 1 GB of RAM and several GB of storage, it can be stated that the BPMN engine has an economical resource consumption. The start-up period of 2.9 s does not differ fundamentally from other Android applications and can be considered tolerable since it is nonrecurring. A high CPU load can be observed during the execution of the activities of the BP. However, a typical BP is mostly in a waiting state. In our example, the health checks are executed every 30 or 60 s. A more efficient set-up could be to wake up the BP not by timer events but by external events, for instance, if a sensor measures a critical value.

We performed the experiments with a simple BP to be able to estimate the resources the BPMN engine requires itself. The resource consumption rises with an increasing number of activities and with more complex tasks, but this depends on the concrete scenario and the requirements of the domain expert. The duration of waiting states and further factors such as the number of service invocations and the quality of the network connectivity also affect the resource consumption.

In summary, it should be possible for a domain expert to create energy-aware BPs with this BPMN engine. Further optimizations can be achieved by switching off certain services of Activiti that may be irrelevant for the process execution and by incorporating Android's Power Manager[29] to improve the energy consumption during waiting states.

With regard to MQTT for Android systems, we refer to IBM's presentation [31] on MQTT in 2013. They measured the exact power consumption when using 3G and Wi-Fi, and compared the overhead of MQTT with that of HTTP.

7. Related work

The advent of resource-efficient WSs has led the SOA research community towards deploying them on embedded devices and sensor networks [7,5,11,14]. Spiess et al. [7] highlight the limitations of BPEL language to meet the dynamic conditions of harsh IoT environments. However, due to verbose SOAP messages such solutions are often too heavy for IoT devices with limited capabilities [20,6]. N. Glombitza et al. [13] address this issue by compressing the large SOAP messages and introducing a *lean transport protocol* to exchange such messages. Instead of SOAP based messaging, we exploit the REST communication paradigm.

Within the same SOA-based approaches, the technical goal of the architecture designed within the COBIs[30] project is to define a set of IoT services that collaboratively provide the functionality required by the BP. While they focus more on semantic interoperability of IoT services, we stress the seamless syntactic interoperability into both resource and service-oriented platforms. The SOCRADES[31] project [7,11] targets resource-constrained devices to participate in service orchestrations based on the WS-* family of standards and a gateway/service-mediator. The project provides both device and application level middleware solutions that are applicable only on SOA based platforms whereas our approach exploits REST as a communication technology that leverages REST-based WSs as a transport; this also fills the gap between SOA and REST paradigms when it comes to the integration of IoT systems. Recently, the makeSense[32] project [32,10] has aimed to solve the integration problem of smart devices and BPs through a novel programming model and language. Similarly, Caracas

29 http://developer.android.com/reference/android/os/PowerManager.html.
30 www.cobis-online.de.
31 www.socrades.eu.
32 http://www.project-makesense.eu/.

et al. [18,19] earlier showed how a BPMN process can be compiled into native source code for sensor devices, without losing too much performance compared to hand-written code. The main goal of such transformations is to use a top down approach to close the gap between models and executable IoT aware processes for a model-driven development. Unlike this, we use a bottom up approach and expose the functionality of IoT devices as compose-able services for standard BPMN-based process.

Within the REST-based approaches, Guinard et al. [16] propose the use of an intermediate gateway that can offer a unified RESTful API to devices by hiding the actual protocols to communicate with them. In the same direction, [21] addresses the issue of how to model and describe the data and functionality provided by services within smart environments to facilitate composition and user-centric interaction with IoT systems. While these two approaches facilitate simple mash-up applications, our focus is to provide a standard compliant integration of smart devices into the BP applications by focusing on events management, dynamic service replacement, and decentralized execution of the IoT aware BP.

In the context of BPMN processes distribution, Hackmann et al. [33] implement the Silver lightweight execution engine for BPEL processes on mobile devices. It supports the core feature set of BPEL, while advanced features like event handling are missing. Ishikawa et al. [34] propose mobile agents that implement workflows. These agents are able to move to WS locations to access them efficiently. In our use case, they could migrate from the desktop computer to the smart phone as soon as the user leaves home. The benefit of migration is that the domain expert does not have to implement the BP twice. In fact, the BPMN engine we are using provides an API that simplifies considerably the implementation of such an approach. However, we did not opt for it, as we believe that context-sensitive BPs running on mobile devices differ from those executed on stationary desktop computers or servers. While stationary computers are immovable and have a rather unvarying environment, mobile devices are exposed to permanently changing surroundings. In case of the migration approach, both the desktop computer and the mobile device would execute the same BP. An adapted version for the mobile device would not be possible.

In [35], Giner et al. introduce their software Presto that supports users to execute their tasks by using the Presto task manager running on their mobile devices. The mobile device may interact with components surrounding the user. In their approach, the BP itself is not necessarily running on the mobile device, rather it sends the pending tasks to the Presto platform on the mobile device. However, we decided to process the sensor data in a BP on the smartphone as it reduces the communication cost considerably. Moreover, it is more robust to missing communication links as the mobile device does not require to constantly communicate with the central BP.

8. Conclusion and future work

In this article, we have presented a REST inspired architecture for interconnecting BPs with IoT resources. We discussed the feasibility of such an integration with the help of an application scenario and documented the main requirements for the architecture based solution that we proposed. The core components of the proposed architecture are briefly described along with their implementation details. Achieved results demonstrate the feasibility of our framework to be deployed for future IoT systems. Therefore, with the introduction of our platform architecture, we intend to provide a base platform on which more advanced features could be designed and built. For example, we are interested in extending Device Status Monitor component to not only report hardware failures but also detect sensor data (incorrect data) or software failures. We also aim at making the proxy based service replacement more scalable to cope with frequent service failures within varying number of composed services.

References

[1] L. Atzori, A. Iera, G. Morabito, The internet of things: A survey, Comput. Netw. 54 (15) (2010) 2787–2805.
[2] D. Yazar, A. Dunkels, Efficient application integration in IP-based sensor networks, in: Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings, BuildSys'09, ACM, New York, NY, USA, 2009, pp. 43–48.
[3] Z. Shelby, C. Bormann, 6LoWPAN: The Wireless Embedded Internet, Wiley Publishing, 2010.
[4] Z. Shelby, Embedded web services, Wirel. Commun. 17 (6) (2010) 52–57.
[5] N.B. Priyantha, A. Kansal, M. Goraczko, F. Zhao, Tiny web services: design and implementation of interoperable and evolvable sensor networks, in: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, SenSys'08, ACM, New York, NY, USA, 2008, pp. 253–266.
[6] A. Caracas, From business process models to pervasive applications: Synchronization and optimization, in: Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on, 2012, pp. 320–325.
[7] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, D. Savio, Interacting with the SOA-based internet of things: discovery, query, selection, and on-demand provisioning of web services, IEEE Trans. Serv. Comput. 3 (3) (2010) 223–235.
[8] S. Mahlknecht, S. Madani, On architecture of low power wireless sensor networks for container tracking and monitoring applications, in: Industrial Informatics, 2007 5th IEEE International Conference on, Vol. 1, 2007, pp. 353–358.
[9] W. Yao, C.-H. Chu, Z. Li, The adoption and implementation of RFID technologies in healthcare: A literature review, J. Med. Syst. 36 (6) (2012) 3507–3525.
[10] S. Tranquillini, et al., Process-based design and integration of wireless sensor network applications, in: A. Barros, A. Gal, E. Kindler (Eds.), Business Process Management, in: Lecture Notes in Computer Science, vol. 7481, Springer, Berlin, Heidelberg, 2012, pp. 134–149.
[11] P. Spiess, S. Karnouskos, D. Guinard, D. Savio, O. Baecker, L.M.S. de Souza, V. Trifa, SOA-based integration of the internet of things in enterprise services, in: 2013 IEEE 20th International Conference on Web Services, Vol. 0, 2009, pp. 968–975.
[12] S. Haller, S. Karnouskos, C. Schroth, The Internet of Things in an Enterprise Context, in: Future Internet - FIS 2008, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 14–28 (Chapter).
[13] N. Glombitza, S. Ebers, D. Pfisterer, S. Fischer, Using BPEL to realize business processes for an internet of things, in: Ad-hoc, Mobile, and Wireless Networks, in: Lecture Notes in Computer Science, Vol. 6811, Springer, Berlin, Heidelberg, 2011, pp. 294–307.
[14] E. Avilés-Lpez, J. Garca-Macas, TinySOA: A service-oriented architecture for wireless sensor networks, Serv. Oriented Comput. Appl. 3 (2) (2009) 99–108.
[15] R.T. Fielding, Architectural Styles and the Design of Network-based Software Architectures (Ph.D. thesis), University of California, Irvine, 2000.

[16] D. Guinard, V. Trifa, E. Wilde, A resource oriented architecture for the web of things, in: Internet of Things (IoT), Vol. 2010, 2010, pp. 1–8.
[17] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, K. Kim, TinyREST - A protocol for Integrating Sensor Networks into the Internet, in: Proc. of REALWSN, 2005.
[18] A. Caracas, A. Bernauer, Compiling business process models for sensor networks, in: Distributed Computing in Sensor Systems and Workshops (DCOSS), Vol. 2011, 2011, pp. 1–8.
[19] A. Caracaay, T. Kramp, On the expressiveness of BPMN for modeling wireless sensor networks applications, in: R. Dijkman, J. Hofstetter, J. Koehler (Eds.), Business Process Model and Notation, in: Lecture Notes in Business Information Processing, vol. 95, Springer, Berlin, Heidelberg, 2011, pp. 16–30.
[20] D. Guinard, V. Trifa, T. Pham, O. Liechti, Towards physical mashups in the web of things, in: Networked Sensing Systems (INSS), 2009 Sixth International Conference on, 2009, pp. 1–4.
[21] E. Aviles-Lopez, J.A. Garcia-Macias, Mashing up the internet of things: a framework for smart environments, EURASIP J. Wirel. Commun. Netw. 2012 (1) (2012) 79.
[22] S. De, P. Barnaghi, M. Bauer, S. Meissner, Service modelling for the internet of things, in: Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on, 2011, pp. 949–955.
[23] M. Kovatsch, S. Duquennoy, A. Dunkels, A low-power CoAP for contiki, in: Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on Mobile Ad-Hoc and Sensor Systems, 2011.
[24] F. Leymann, BPEL vs. BPMN 2.0: Should you care? in: Business Process Modeling Notation, Springer, 2011, pp. 8–13.
[25] M. Thorpe, C. Ke, Simple Rule Markup Language (SRML): A general XML rule representation for forward-chaining rules, XML coverpages, 2001, p. 1.
[26] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, et al. Swrl: A semantic web rule language combining owl and ruleml, W3C Member submission, Vol. 21, 2004, p. 79.
[27] P. Browne, JBoss Drools Business Rules, Packt Publishing Ltd, 2009.
[28] M. Zur Muehlen, M. Indulska, Modeling languages for business processes and business rules: A representational analysis, Inf. Syst. 35 (4) (2010) 379–390.
[29] C.L. Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem, Artificial Intelligence 19 (1) (1982) 17–37.
[30] D. Locke, MQ Telemetry Transport (MQTT) V3. 1 Protocol Specification, IBM developer Works Technical Library.
[31] H. Sjöstrand, Low Latency Mobile Messaging using MQTT, IBM Impact 2013 (April 2013).
[32] F. Casati, F. Daniel, et al. Towards business processes orchestrating the physical enterprise with wireless sensor networks, in: Software Engineering (ICSE), 2012 34th International Conference on, 2012, pp. 1357–1360.
[33] G. Hackmann, M. Haitjema, C. Gill, G.-C. Roman, Sliver: A BPEL workflow process execution engine for mobile devices, in: Service-Oriented Computing–ICSOC 2006, Springer, 2006, pp. 503–508.
[34] F. Ishikawa, N. Yoshioka, S. Honiden, Mobile agent system for Web service integration in pervasive network, Syst. Comput. Japan 36 (11) (2005) 34–48.
[35] P. Giner, C. Cetina, J. Fons, V. Pelechano, Developing mobile workflow support in the internet of things, IEEE Pervas. Comput. 9 (2) (2010) 18–26.