Interacting with the SOA-based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services

Dominique Guinard, *Student Member, IEEE,* Vlad Trifa, *Student Member, IEEE,*
Stamatis Karnouskos, *Senior Member, IEEE,* Patrik Spiess, *Member, IEEE,*
and Domnic Savio, *Member, IEEE*

Abstract—The increasing usage of smart embedded devices in business blurs the line between the virtual and real worlds. This creates new opportunities to build applications that better integrate real-time state of the physical world and hence provide enterprise services that are highly dynamic, more diverse and efficient. Service Oriented Architecture approaches traditionally used to couple functionality of heavyweight corporate IT systems, are becoming applicable to embedded real-world devices, i.e. objects of the physical world that feature embedded processing and communication. In such infrastructures, composed of large numbers of networked, resource-limited devices, the discovery of services and on-demand provisioning of missing functionality is a significant challenge. We propose a process and a suitable system architecture that enables developers and business process designers to dynamically query, select, and use running instances of real-world services (i.e. services running on physical devices) or even deploy new ones on-demand, all in the context of composite, real-world business applications.

Index Terms—Service-oriented Architecture (SOA), Service Discovery, Web Services, REST, Web of Things, Device Integration, Composite Applications, Wireless Sensor (Actuator) Networks, Context modeling, Ubiquitous Business processes.

1 Introduction

The last years, we have witnessed two major trends in the world of embedded devices. Firstly, hardware is becoming smaller, cheaper, and more powerful. According to the Internet of Things vision (IoT) [1], the majority of the devices will soon have communication and computation capabilities, which they will use to connect, interact, and cooperate with their surrounding environment. Secondly, the software industry is moving towards service-oriented integration technologies. Especially in the business software domain, complex applications based on the composition and collaboration among diverse services have been appearing. The Internet of Services vision (IoS) [2] assumes this on a large scale, where services reside in different layers of the enterprise *e.g.* different operational units, IT networks, or even running directly on devices and machines within the company. As both of these trends are not domain specific but common to multiple industries, we are facing a trend where the service-based information systems blur the border between the physical and virtual worlds, providing a fertile ground for a new breed of real-world aware applications. The efficiency of such applications will heavily depend on the cooperation of heterogeneous networked embedded devices among themselves and with business systems [3]. In this Internet of Things, we expect that dynamic network discovery, query, selection, and on-demand provisioning of Web services will be of crucial importance.

In the future Internet, real-world devices will be able to offer their functionality via SOAP-based Web Services (WS-*) or RESTful APIs [4], enabling other components to interact with them dynamically. The functionality offered by these devices (*e.g.* the provisioning of online sensor data) is often referred to as *real-world services* because they are provided by embedded systems that are related directly to the physical world. Unlike traditional enterprise services and applications, which are mainly virtual entities, real-world services provide real-time (we refer to relatively low-latency, not necessarily offering hard real-time guarantees) data about the physical world. Armed with this additional knowledge, one can support a more efficient decision taking process. Hence, devices providing their functionality as a Web services can be used by other entities such as enterprise applications or even other devices. No device drivers are needed anymore and a new level of efficiency can be achieved as Web service clients can be generated dynamically at run-time.

Trends show that in the future, a much more diversified infrastructure will emerge, and the way we interact with it will change accordingly. As depicted in Figure 1, mash-ups of services will be created and used across

- *D. Guinard and Vlad Trifa are with SAP Research Zurich and the Institute for Pervasive Computing, ETH Zurich, Zurich, Switzerland. E-mail: see www.guinard.org*
- *S. Karnouskos, P. Spiess and D. Savio are with SAP Research Karlsruhe, Germany.*

Manuscript received April 30, 2009; revised November 27, 2009.

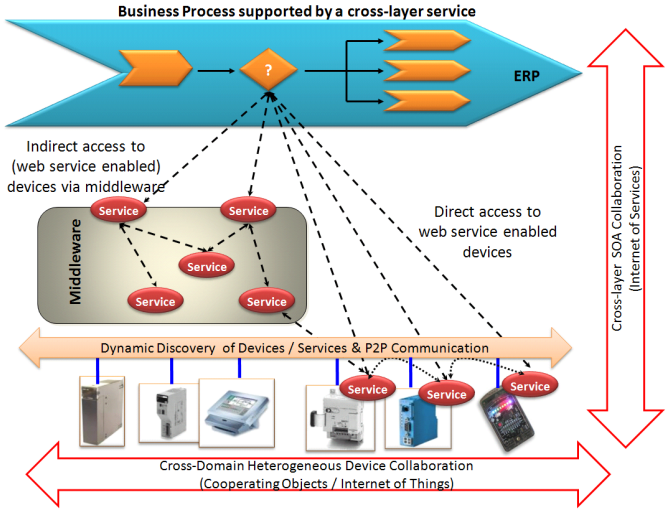

Figure 1. The Collaborative Future Internet Vision

various system layers. We will experience horizontal collaboration directly between devices with no human intervention [3], but also vertical collaboration between devices and online services, applications, and people. Enterprise applications will be able to connect directly to devices with no need for proprietary drivers, while non Web service-enabled devices can still be used by wrapping their functionality behind gateways. Peer-to-peer communication among devices will push services down to the device layer and create new opportunities for functionality discovery and collaboration.

According to OnWorld [5], the global market for Wireless Sensor Network (WSN) systems and services is expected to skyrocket to about $4.6 billion in 2011, up from approximately $500 million in 2005. There will be a worldwide market of $5.3 billion (conservative estimate) for the industrial control segment only, comprising 4.1 million nodes by 2010. OnWorld's most aggressive forecast for all wireless sensor (& control) network segments is $8.2 billion by 2010, comprising 184 million deployed nodes. Thus, the business opportunities for real-world services are promising. Even if a fraction of this holds true for the near term, we will witness a mass market penetration of networked embedded devices. Services taking advantage of the unprecedented ease of consumption of device functionality will give birth to new innovative applications and provide both revenue generating and cost saving business advantages. From a technology point of view, the key challenge is how to discover, assess, and efficiently integrate the real-world services into business applications.

1.1 Background and Related Work

Several efforts have explored the integration of real-world and enterprise services e.g. [6, 7]. However, the protocols used do not offer uniform interfaces across the application space and are too complex to integrate with traditional enterprise applications. To ensure interoperability across all systems, recent work has focused on applying the concept of Service-oriented Architecture (SOA), in particular Web Service standards (SOAP, WSDL, etc.) directly on devices (e.g. [8–10]). Implementing WS-* standards on devices presents several advantages in terms of end-to-end integration and programmability, by reducing the need for gateways and translation between the components. This enables the direct orchestration of services running on devices, e.g. sensors monitoring the temperature of shipments, with high-level enterprise services, e.g. offered by an Enterprise Resource Planning (ERP) application.

Embedding SOA concepts at device level initially seems a good idea, however we have to keep in mind that SOA standards were designed primarily for connecting, complex, and rather static enterprise services. Thus implementing WS-* standards directly on devices is not always straightforward. Unlike enterprise services, real-world services are deployed on resource constrained devices, e.g. with limited computing, communication and storage capabilities. This requires significant simplification, optimization, and adaptation of SOA tools and standards [10]. Additionally, real-world services are found in highly dynamic environments where devices and their underlying services constantly degrade, vanish, and possibly re-appear. As such, this infrastructure can not be considered as static and long-lived as traditional enterprise services. This implies the need for automated, immediate (dynamic) discovery of devices and services as well as their effective management.

A crucial **challenge** for SOA developers and process designers is to find adequate services for solving a particular task [11]. This process is often referred to as "service discovery" or simply "discovery" [12] and is end-user driven. We shall distinguish it from the "network discovery" [7] of services which is machine driven and occurs at the network level. Discovering enterprise services often implies manually querying a number of registries, such as Universal Description and Discovery and Integration (UDDI) registries, and the results depend largely on the quality of the data within that registry. While such an approach is adequate for a rarely changing set of large-scale services, the same is insufficient for the requirements of the dynamic real-world services. Registering a service with one or more UDDIs is rather complex, and does not comply with the minimization of usage of the devices' limited resources. Furthermore extensive description information is necessary [13], while the device can only report basic information about itself and the services it hosts. Trying to reduce the complexity of registration and discovery, different research directions have been followed in order to provide alternatives or extensions of the UDDI standard [11, 12, 14]. However, also these do not take into account the particular requirements of real-world services.

1.2 Our Contributions

Based on our experiences within SAP, in developing real-world services for the enterprises, we introduce here a set of requirements to facilitate the querying and discovery of real-world services from enterprise applications:

1) **R1: Minimal Service Overhead**. As most real-world services are offered by embedded devices with (very) limited computing capabilities there is a need for a lightweight service-oriented paradigm which does not generate too much overhead compared to using functionality through the proprietary APIs.
2) **R2: Minimal Registration Effort**. A device should be able to advertise its services to an open registry using network discovery. The process should be "plug and play", without requiring human intervention. A device should also be expected to provide only a small amount of information when registering.
3) **R3: Support for Dynamic and Contextual Search**. It should be possible to use external sources of information to better formulate queries. Furthermore, the queries should go beyond simple keyword search and take into account user-quality parameters such as context (*e.g.* location, Quality of Service (QoS), application context). Support for context is essential as the functionality of most real-world devices is task-specific within a well-defined context (*e.g.* a building, a manufacturing plant, etc.).
4) **R4: Support for On-Demand Provisioning**. Services on embedded devices offer rather atomic operations, such as obtaining data from a temperature sensor. Thus, while the WSN (Wireless Sensor Networks) platforms are rather heterogeneous, the services that the sensor nodes can offer share significant similarities and could be (re)deployed on-demand per developer request.

In the work presented here we build upon existing research on device integration through services [8–10, 15]. Our key **contribution** is the **service discovery process for real-world services** initially introduced in [16]. This process is shown on Figure 4 and described in detail in Section 4. The goal of this process, called Real-World Service Discovery and Provisioning Process (RSDPP), is to **assist the developers at development time in the discovery of real-world services** to be included in composite applications. This innovative process fulfills the requirements of real-world services we described (R1 to R4) above as follows: In order to ensure a minimal overhead (R1) for providing the functionality of embedded devices as service we propose two approaches. In the first one we use the Device Profile for Web Services (DPWS) [8] and its dynamic network discovery mechanism. DPWS defines a limited set of WS-* standards which are implementable on relatively resource-constrained devices. We will describe DPWS in Section 3.1.

As an alternative to fulfill requirement R1 we also introduce the design of Resource Oriented real-world devices, that is embedded devices providing their functionality through a RESTful API [4, 17, 18]. REST (Representational State Transfer) [19] is the architectural principle that lies at the heart of the Web and shares a similar goal with DPWS, which is to increase interoperability for a looser coupling between the parts of distributed applications [20] towards serendipitous re-use of services. We further describe the concept of resource-oriented real-world devices in Section 3.2.

The minimal registration effort (R2) requirement is met by using DPWS [8] and its network discovery mechanism. In Section 3.2 we also describe how Resource Oriented devices can also fulfill this requirement.

We further ensure the minimal registration (R2) effort and support for dynamic search (R3) by extending developer provided keywords with vocabularies of related terms also known as "lightweight ontologies" [21]. We generate these terms dynamically, by using semi-structured Web resources like Wikipedia and Yahoo! Web Search. This part of the process called Query Augmentation, is described in Section 4.1.

The dynamic search requirement (R3) is also fulfilled by taking into account the developer context and matching it with the extracted context of real-world services. This developer quality information is then used for adequate service selection when retrieving and ranking services as explained in Section 4.2.

The requirement for on-demand dynamic provisioning (R4) is fulfilled by a software architecture that enables the developer to automatically deploy services on devices when no requirements-satisfying service was found in the environment [22]. This architecture is described in Section 4.3.

Finally, we present our implementation within an enterprise application (based on Java Enterprise Edition and SAP NetWeaver) as well as its deployment and validation of our results in Section 5.

Before describing the process itself we start with an overview of the framework in which the RSDPP was developed and in particular on how devices can register themselves and advertise their services in an automated manner.

2 The SOCRADES Integration Architecture

The process described in this article has been developed and implemented as part of the SOCRADES Integration Architecture (SIA) [9, 23, 24], which is depicted in Figure 2. The role of SIA is to enable the ubiquitous integration of real-world services running on embedded devices with enterprise services. WS-* Web Service standards constitute the *de facto* communication method used by the components of enterprise-level applications, and for this reason SIA is fully based on them. In this manner,

business applications can access near real-time data from a wide range of networked devices through a high-level, abstract interface based on Web Services. Furthermore, the SIA also supports RESTful services in order to be able to communicate with many emerging Web 2.0 services. This enables any networked device that is connected to the SIA to directly participate in business processes while neither requiring the process modeler, nor the process execution engine to know about the exact details of the underlying hardware.

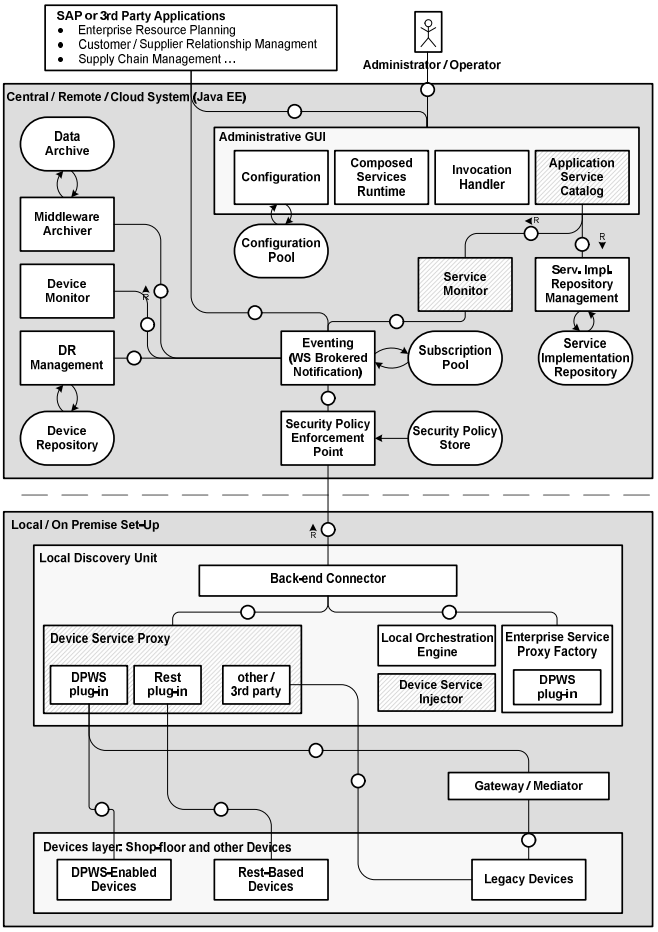

Figure 2. The SOCRADES Integration Architecture (SIA)

The details of SIA can be found in [24], therefore here we focus those components of the architecture, which are relevant for RSDPP. The components with gray, diagonal patterns play a key role in the process. SIA is split in two parts: A "local, on premise" part which features a **Local Discovery Unit (LDU)** and is running at the local network that contains the devices to be integrated, and a central system (anywhere on the network or even Internet) that hosts enterprise-level applications. Although the quantitative relation between both subsystems is $m : n$, in a typical single-site setup there will be only one central system and one or more on-premise systems. In a multi-enterprise collaborative landscape however, we would witness several "central" systems collaborating and possibly various "local" systems reporting to more than one "central" ones.

In the local subsystem at Device Layer there are several embedded devices that are running various services. SIA is able to interact with devices using several communication protocols, such as DPWS, OPC-UA, REST, etc., however in this article we focus solely on DPWS and REST-enabled devices. Since DPWS-enabled devices support Web Services, they also can bypass SIA for a direct connection to Enterprise Applications, which is desirable in some use cases, but not for the majority of foreseen ones. Furthermore, SIA allows applications to subscribe to any events sent by the devices, offering a publish/subscribe component that supports WS-Notifications. It also offers buffered invocations of hosted services on devices that are only intermittently connected, by receiving notifications when the device becomes available again or having the system cache the message and delivering it when the device is ready to receive it. As such SIA is a vital component hiding and managing the complexity of real-world landscapes from the enterprise service developers, easing their tasks.

The key component that connects the local subsystem with the central one is the local network discovery unit (or LDU). The LDU module **Device Service Proxy** scans the local network for DPWS and REST devices and reports their connecting and disconnecting to the central system. It acts as an intermediary that provides uniform access to different classes of devices through a set of platform-dependent plug-ins. Some of the advanced features of the LDU are a lightweight local orchestration engine that allows for autonomous execution of local processes, a **Device Service Injector** that is able to change the embedded software on devices in order to (un)deploy or (re)configure embedded services (used for on-demand provisioning as in Section 4.3.2), and an enterprise service proxy factory that can make services from a business application available in the local network so that devices can access those back-end services through the same protocols as they would use to communicate with other devices. All these are realized by (unidirectional) Web service calls from the LDU to the central system, therefore allowing for firewall-friendly operations and operation through an HTTP(s) proxy.

In the central subsystem, we have implemented higher-level components to ease the management and use of devices in a standardized and uniform way. The **Device Repository** holds all dynamically acquired but mostly static device information (metadata) of all on-line and off-line devices (of all connected local subsystems), while the **Device Monitor** contains information about the current state of each device. The Device Monitor acts as the single access point where enterprise applications can find all devices even when they have no direct access to the shop floor network or such access is not wished (*e.g.* flooding of shop-floor with network discovery messages).

At the same time, information about the different

services hosted on the device (typically described using WSDLs) will be retrieved and forwarded using an event to the **Service Type Repository** and **Service Monitor** as shown on Figure 4. The former only contains information about the service types without their respective endpoint references; the latter contains information about the available service instances hosted by the devices and their endpoint references, and also installable service types. The Service Type Repository acts as a facade for querying the underlying repositories and monitors for pointers to running service instances.

The RSDPP process is mainly orchestrated from the component called **Application Service Catalog**. It contains a GUI, hosts and applies the service instance ranking strategies (see Section 4.2.3), and controls the interaction with the other components involved in the process.

3 Network Discovery of Embedded Devices

Along with increasingly dynamic infrastructures where mobile devices appear or disappear from the network at operation time, there is a strong need for tools to simplify the management and interconnection of networked devices. Network discovery is a central process in ubiquitous and distributed computing [7]. In contrast to the user-oriented discovery, network discovery enables *machines* to automatically register themselves and advertise their services on the network. In a way, network discovery is the bootstrap of service discovery for end-users. In this field, many protocols have been proposed such as the Service Location Protocol (SLP), Universal Plug and Play (UPnP), Device Profile for Web Services, Sun's Jini, or Apple's Bonjour.

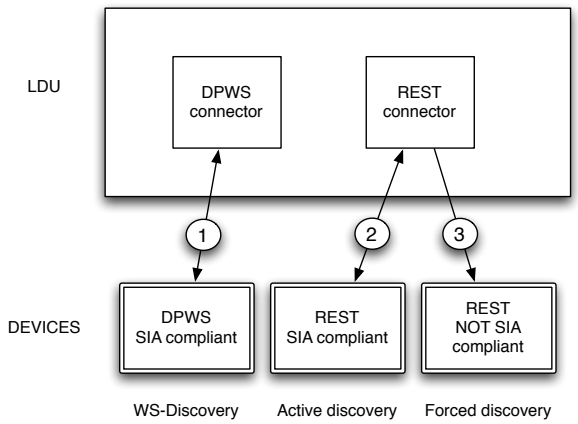

Figure 3. Three Alternative Mechanisms for Network Discovery of Real-World Services Hosted on Physical Devices

Such a discovery mechanism is essential in scenarios where devices can join the network and discover dynamically the services offered by other devices, and will be unavoidable requirement in future enterprise scenarios with dynamic and adaptive production lines. In this Section, we present three alternative mechanisms we have used in our system for network discovery of devices. As depicted in Figure 3, the process of finding out real world services running on physical devices can be done as follows:

1) WS-Discovery on which DPWS is based (both active and passive).
2) RESTful active network discovery, where a device notifies its presence to the LDU automatically.
3) Passive RESTful discovery for REST-enabled devices that do not comply with SIA network discovery. Passive RESTful discovery is triggered by passing the URI of the device to be registered in the system as a parameter of the forced discovery method call.

3.1 DPWS Web Services

Device Profiles for Web Services (DPWS) is a subset of Web service standards (such as WSDL and SOAP) that allows minimal interaction with Web services running on embedded devices. DPWS is the successor of Universal Plug and Play (UPnP) as in essence it specifies a protocol for seamless interaction with the services offered by different embedded devices. However DPWS is fully aligned with Web Services technologies. The various specifications DPWS include support for (secure) messaging, service discovery and description, and eventing for resource-constrained devices. Devices can run two types of services: a) hosting services and b) hosted services. *Hosting services* are directly related to DPWS and support the low-level, generic (meta-)services such as **network discovery services** used by a device connected to a network to advertise itself and to discover other devices, **metadata exchange services** to provide information about a device and the hosted services on it, and **asynchronous publish and subscribe eventing**, allowing to subscribe to asynchronous event messages produced by a given hosted service. *Hosted services* are mostly functional and depend on their hosting device and its functionality for network discovery.

As services run directly on limited networked devices, a robust mechanism is needed, in order to find new devices as they connect to the network, and dynamically retrieve metadata about it and the services it hosts. To achieve this, the WS-Discovery specification is used. When a new device joins the network, it will multicast a HELLO message via the UDP protocol. By listening to this message, clients can detect new devices and in a second step retrieve their metadata. This in turn triggers the sending of an appropriate message to the SIA Device Monitor, containing the device's static metadata. The metadata information can be classified into a certain set of metadata *classes* (see **categories** paragraph below), and is required for searching services according to more detailed criteria. This data about the device is stored by the higher units for future usage.

Filtering information can be included in a *Probe* message sent to a multicast group. Devices whose metadata match the probes' content will send a *ProbeMatch* response directly to the client (in unicast mode). Similarly, to locate a device by name, a client sends a *Resolve* message to the same multicast group and the device that matches sends a *ResolveMatch* response directly to the client. After this network-level scan, the result set can be further narrowed by matching keywords or textual information that describe both static (device type, available sensors on board) and dynamic properties of devices (QoS, physical location, available battery life, network connectivity, or available sensors).

Metadata categories The DPWS metadata of devices and services can be classified in different categories, as follows:

- **Scopes** a set of attributes that may be used to organize devices into logical or hierarchical groups, *e.g.* according to their location or access rights.
- **Model and Device metadata** provides information about the type of the device like manufacturer name, model name, model number, etc. as well as information on the device itself such as serial number, firmware version and friendly name.
- **Types** are a set of messages the device can send and/or receive; these can be either functional WSDL port types (*e.g. 'turn on'*, *'turn off'*) or abstract types grouping several port types and/or hosted services (*e.g. 'printer'*, *'lighting'*, *'residential gateway'*).
- Links to **WSDL document** (i.e. URLs), containing the port types (operations and message structures) implemented and the endpoint of hosted services.

3.2 RESTful Services for the Real-World

The architectural principles that are core to the Web, namely Representational State Transfer (REST) as defined by Roy Fielding [19], share a similar goal with WS-* Web services, which is to increase interoperability for a looser coupling between the parts of distributed applications [20]. However, the goal of REST is to achieve this in a more lightweight and simpler manner seamlessly integrated to the Web. REST uses URIs for encapsulating and identifying services on the Web. In its Web implementation it also uses HTTP as a true application protocol. This way, REST brings services "into the browser": resources can be linked, bookmarked, cached, searched for, and the results are directly visible within any Web browser. Requests for services (i.e. verbs on resources) are formulated using a standard URI. For instance, typing a URI such as `http://.../spot1/sensors/temperature` in a browser, can be used to request the resource (here: operation) "temperature" of the resource "sensor" of "spot1" with the verb GET HTTP method.

Traditionally, REST has been used to integrate websites together. However, the lightweight and ubiquitous aspects of REST makes it an ideal candidate to build an "universal" API (Application Programming Interface) for embedded devices. This concept is often referred to as "Web of Things" [17, 18, 25].

Since many such devices usually offer rather simple and atomic functionalities (for example reading sensor values), modeling them using REST is often straightforward. While REST services are well adapted for rather atomic services, thus cover a fair part of the basic services offered by embedded devices, they have limitations when modeling services which require complex input and/or deliver complex outputs. According to our own experience and of others [26], in traditional integration patterns based on WS-* Web Services, we suggest that WS-* services are to be preferred for highly complex real-world integration and rather static use-cases, such as those involving complex business processes or those requiring high reliability or security, for example composing a manufacturing process on several machines. For lightweight and more end-user oriented applications, the RESTful approach offers significant advantages such as simplicity, direct Web integration and looser-coupling [4, 26]. As both scenarios are needed for truly flexible enterprise applications, the Local Discovery Unit of our solution supports both DPWS and REST-enabled devices.

HTTP has been designed as a high-level application protocol, therefore the notion of network discovery is not part of HTTP specifications. In the modern Web, resources are discovered by following outgoing links from each resource, but this model requires to know beforehand the URI of a boot-strap resource, therefore is not suited for discovering new devices that appear on the network. To counter that, mRDP [27] proposed as a simple HTTP-based semantic resource discovery mechanism, based on UDP broadcasting. In our approach, we implemented alongside WS-Discovery, a RESTful network discovery mechanism suited for devices that do not support DPWS, but only HTTP. In the passive network discovery of REST-enabled devices, each device must announce itself using a HELLO message (exactly as in the DPWS network discovery case), however, upon reception of the acknowledgment from the LDU, the device will generate a PUT request on the LDU REST server, and register itself into the LDU using a predefined device registration procedure.

Since there is no standard for the discovery of RESTful services, we cannot expect any RESTful device to be compliant with the custom HTTP passive discovery protocol we have just described. In this case, we force the network discovery (active) upon the device, by issuing a GET request on the device page which needs to be known. The LDU will parse the device page and retrieve the necessary information from the device and its services. The device page is a machine-readable "RESTful API" described in HTML. To enhance this description we experimented using metadata contained in a specific microformat[1] we have developed specifically for devices.

1. www.microformats.org

The metadata contained in this microformat is the same as the metadata used in the network discovery process of DPWS devices so that the same information is contained in the service repositories of the architecture for both REST and DPWS services.

4 Real-World Service Discovery and Provisioning Process (RSDPP)

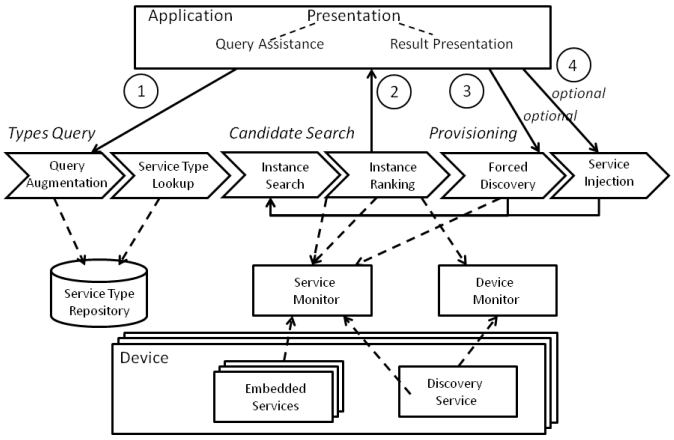

Figure 4. Overview of the Real-World Service Discovery and Provisioning Process (RSDPP).

After describing the way devices and their services are advertised, this Section describes the RSDPP and its underlying steps. As illustrated in Figure 4 step 1, the process begins with a **Types Query** after the network discovery of devices has been executed. In this sub-process the developer uses keywords to search for services, as she would search for documents on any search engine. Subsequently this query is extended with related keywords fetched from different websites, and used to retrieve types of services that describe the functionality, but not yet the real-world device it runs on. This is the task of the **Candidate Search**, where the running instances of the service type are retrieved and ranked according to context parameters provided by the developer (Fig 4, step 2). In case no service instance has been found, the process goes on with **Provisioning**. It begins with a forced network discovery of devices, where the devices known to provide the service type the developer is looking for, are asked to acknowledge their presence (step 3). If no suitable device is discovered, a service injection can be requested. In this last step the system tries to find suitable devices that could run the requested service, and installs it remotely (step 4).

4.1 Types Query

In the first part of the discovery process (step 1 on Figure 4), the developer or process designer enters keywords describing the type of service she wants to find (step 1 on Figure 5). A **Service Type** is a generic WSDL file describing the abstract functionalities of a real-world service, but not bound to any particular end-point of a concrete real-world device. The entered keywords are sent to the **Query Augmentation** module which extends the query with additional keywords. The output of this module is then used to retrieve and rank types of services according to user-quality parameters such as the current user context.

4.1.1 Query Augmentation and Assistant

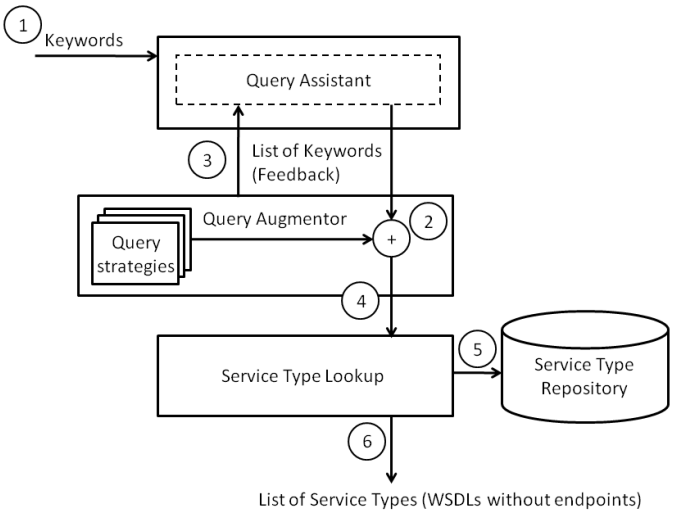

Figure 5. Looking for a Service Type.

In conventional service discovery applications, the keywords entered by the user would be sent to a Service Repository to find types of services corresponding to the keywords. The problem with this simple keyword matching mechanism is that it lacks flexibility. As an example lets assume a developer who wants to find services offered by a *"smart meter"*, a term often used to describe a next generation device that can measure the energy consumption of other devices and possibly control them depending on built-in logic. Typing "smart meter" only, will likely not lead into finding all the corresponding services, because services dealing with energy consumption monitoring might not be tagged with the *"smart meter"* keyword but simply with "electronic meter" etc. We want to avoid the construction of domain ontologies, and to minimize the amount of data that embedded devices need to provide upon network discovery and service registration. Thus, we propose a system that uses services on the Web to extend queries without involving communication with the embedded devices or requiring complex service descriptions from them. This is the query augmentation shown on step 2 of Figure 5.

Our idea is to use existing knowledge repositories such as Web encyclopedias (*e.g.* Wikipedia) and search engines (e.g. Google, Yahoo! Web Search), in order to extract "lightweight ontologies" [21] or vocabularies of terms from their semi-structured results. The basic concept of the query augmentation (step 2 on Figure 5) is

to call $1..n$ Web search engines or encyclopedias with the search terms provided by the developer, for instance "smart meter". The XHTML result page from each Web resource is then automatically downloaded and analyzed. The result is a list of keywords, which frequently appeared on pages related to "smart meter". A number of the resulting keywords are thus related to the initial keyword i.e. "smart meter" and therefore can be used when searching for types of services corresponding to the initial input.

An invocable Web-resource together with several filters and analysis applied to the results is called a *Query Strategy*. The structure is based on the Strategy Pattern [28], which enables us to encapsulate algorithms into entirely independent and interchangeable classes. This eases the implementation of new strategies based on Web resources containing relevant terms for a particular domain. A simplified class diagram of the Query Strategy framework is depicted on Figure 6. Any Query Strategy has to implement the `AbstractStrategy` class which provides the general definition of the algorithm. As an example the `YahooStrategy` is a concrete implementation of this algorithm using the Yahoo! Search service. Furthermore, strategies can have extensions, adding more specific functionality to a concrete instance of a Query Strategy. As an example the `WikipediaStrategy` can be extended with the `WikipediaBacklinks` class. This particular extension is using the backlinks operation offered by Wikipedia in order to know what pages are linking to the currently analyzed page similarly to what the well-known PageRank uses to rank websites [29]. This information is then used by the WikipediaStrategy to browse to related pages and gather relevant keywords. As such, our approach builds on top of existing ranking and connectivity approaches on the Web.

Furthermore, Query Strategies can be combined in order to get a final result that reflects the successive results of calling a number of Web-resources. The resulting list of related keywords is then returned to the developer in the Query Assistant, where she can (optionally) remove keywords that are not relevant (step 3 of Figure 5).

The implementation of the Query Strategy architecture makes it easy to test combinations of several strategies together with their extension. We implemented a number of these, and their evaluation is presented in Section 5.2.

4.1.2 Service Type Lookup

The augmented query is used to determine any matching service types in the Service Type Repository (step 4 and step 5 on Figure 5). All service types that match any of the keywords supplied are found; both those manually entered and those determined automatically by the augmentation step. The query keywords are matched against all metadata of a service type that was sent to the Service Monitor upon network discovery or extended by a manual entry. This includes human readable descriptions, contact information, legal terms, explicit keywords

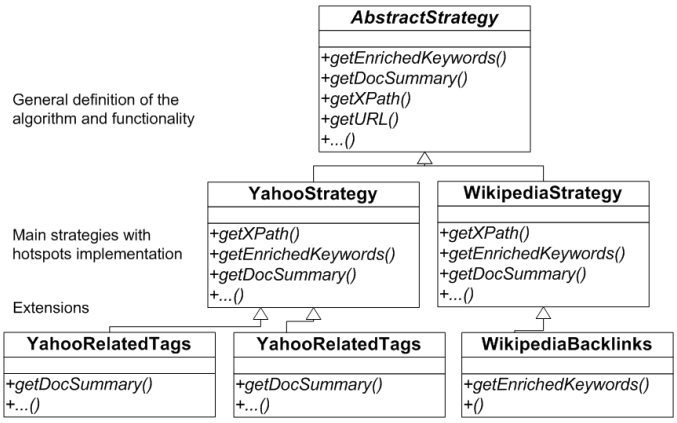

Figure 6. Architectural overview of the Query Strategies based on the Strategy and Template software design patterns.

and interface descriptions (WSDL). Additionally, structured technical metadata is considered, *e.g.* dependency information between service types, and requirements of the service type on underlying hardware. The result of the Service Type Lookup is a list of service types that potentially support the functionality the developer is looking for.

4.2 Candidate Search

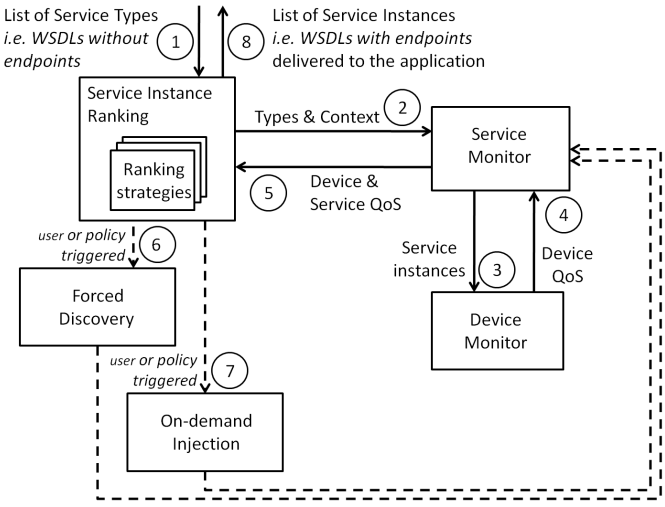

Figure 7. Ranking and optionally Provisioning Service Instances.

Real-world devices are volatile *e.g.* often connecting and disconnecting, thus we need to decouple the discovery of service types from the discovery of actual instances of services. The Candidate Search (step 2 on Figure 4) models the discovery of running service instances. The first step in this sub-process is for the developer to select the suitable types of services by browsing their details (step 1 on Figure 7). Alternatively she can select all the types retrieved in the Types Query part of the process.

4.2.1 Context Extractor

One of the main differences between real-world service and virtual services is that real-world services are directly linked to the physical world. As a consequence, the context in which a service exists as well as the context in which the developer initiates the discovery of a service are highly relevant. Context is information that qualifies the physical world, and it can help in both reducing the number of services returned to the developer, as well as in finding the most appropriate services for the current environment [30].

To satisfy the requirements of real-world service discovery, we propose modeling the context by two distinct parts inspired from [31]: the *digital environment*, which we define as everything that is related to the virtual world the developer is using, and the *physical environment*, which refers to properties of the physical situation the developer currently is located in or wants to discover services about.

The *digital environment* is composed of Application Context and Quality of Service. The **Application Context** describes the business application the developer uses when trying to discover services, *e.g.* the type of application she is currently developing or the language currently set as default. Such information co-determines the services a developer is looking for and can reduce the discovery scope. The **QoS Information** reflects the expectations of the developer (or of the application she is currently using) in terms of how the discovered service is expected to perform. Our current implementation supports service health and network latency, i.e. the current status of the service and the network transmission delay usually measured when calling it.

The *physical environment* is mainly composed of information about location. Developers are likely to be looking for real-world services located at a particular place, unlike when searching most virtual services. We decompose the location into two sub-parts following the Location API for Mobile Devices (as defined in Java Specification Request JSR-179). The **Address** encapsulates the virtual description of the current location, with information such as building name, floor, street, country, etc. and the **Coordinates** are GPS coordinates. In our implementation the location can either be automatically extracted *e.g.* if the developer looks for a real-world service close to her location, or it can be explicitly specified if she wants a service located close to a particular location *e.g.* in a form of radius.

In the RSDPP Context, extraction on the developer side is done at step 2 of Figure 7. It is worth noting that the context on the developer side is meant to reflect the expectations or requirements with regard to the services that are going to be returned. As an example, during this phase the developer can express the wish for a service to be physically close to her current location, or she can quantify the importance of context parameters such as Quality of Service.

This developer-quality information is then going to be compared to the service and device side context by the Service Instance Ranking component (see Section 4.2.3) in order to select and rank the most relevant service instances.

4.2.2 Service Instance Search

In step 3 of Figure 7, the identifiers of the selected service types and the context object extracted on the developer-side are sent to the Service Monitor. This component is the link between service types and running instances of these services. Thanks to the dynamic network discovery of devices (explained in Section 3.1) the Service Monitor and the Device Monitor know what devices are currently providing which service types. In steps 4 and 5 of Figure 7, the Service Monitor queries the Device Monitor for the context of the selected service instances. The digital environment context parameters such as the Quality of Service, are derived by polling the devices from time to time, as well as by monitoring the invocations of services and calculating their execution time.

Getting the context parameters related to the physical environment of a service instance is slightly more complicated. Indeed, as an example it can not be expected from each real-world device to know its location. Thus, we suggest taking a best effort strategy, where each actor of the discovery process is trying to further fill-in the context object. As an example, consider a mobile sensor node without a coordinates-resolving module (*e.g.* a GPS). When discovered by the Local Discovery Unit (see Section 3.1 and Figure 2), the sensor node does not know its location and thus can not fill-in the Address and Coordinates fields of the context object. The LDU however, is a usually immobile component and can be configured at setup time with its location and current address. As a consequence the LDU can fill the Address and Coordinate fields of the sensor node with its own location (within a specific radius). While not entirely accurate with respect to the sensor's exact location, this information will already provide a hint which can be of value to the developer. In the future, more sophisticated methods can be used at device or LDU level (especially if any of them are mobile) *e.g.* [32] in order to acquire location information with the required accuracy. Similarly, since we can not expect every LDU to provide a full contextual profile, the Service Monitor has its own default context component which can again be used to extend the information the device and LDU provided. The final context information is packed into a context object for each device running the selected Service Instances.

If no appropriate service instances have been found, the optional step 7 (described in 4.3) and step 8 (described in 4.3.2) are taken; otherwise the process continues with Service Instance Ranking.

4.2.3 Service Instance Ranking

The Service Instance Ranking component is responsible for sorting the instances according to their compliance

with the context specified by the developer or extracted from her machine. As shown in step 6 of Figure 7, the Service Instance Ranking component receives a number of service instances alongside with their context object. It then uses a Ranking Strategy to sort the list of instances found. For instance, a Ranking Strategy could use the network latency so that the services are listed sorted according to their network latency; another could rank instances according to their compliance with the current location of a developer or the target location she provided.

As for Query Strategies (see Section 4.1.1), Ranking strategies can be well modeled using the Strategy pattern. In this way, new strategies can be easily implemented and integrated. Furthermore, we extend the pattern to support chained ranking strategies, in order for the resulting ranking to reflect a multi-criteria evaluation. Each ranking criterion can use both the context information of the instances gathered during the Service Instance Search, and the context information extracted on the developer side in step 2 of Figure 7. Thus, instances can be ranked against each other and/or against the context of the developer (*e.g.* her location). The output of the ranking process is an ordered list of running service instances corresponding both to the extended keywords and to the requirements in terms of context expressed by the developer. The pattern-based design of the component makes it possible to extend the strategies to accommodate emerging research *e.g.* [33–35] on matching and ranking that needs however to be adapted to consider the specifics of real-world services.

4.3 On-Demand Service Provisioning

In case no running service instance has been found, On-Demand Service Provisioning will first actively try to discover service instance on the network that matches the developer's requirements. If this also fails, installation of services on suitable devices will be carried out.

4.3.1 Forced Network Discovery of Devices

As discussed in Section 3.1, passive DPWS network discovery can be unreliable depending on the mechanism used. This is due to the fact that UDP is used, which provides an unreliable service where datagrams may arrive out of order, appear duplicated, or simply get lost without notification. Furthermore, this mechanism might take a long time to propagate across the whole system (especially when we have thousands of on-device services *e.g.* [36]) as UDP packets are multicasted only in local networks. When up-to-date information is needed, the *forced* network discovery mechanism can be used, particularly within dynamic environments where devices with unknown capabilities continuously connect to or disconnect from the network. This dynamic process can use different types of filters to specify the device type and the scope, as well as other additional semantic information. This is useful to restrict the result set when looking for new devices, as only devices matching the specified criteria will respond. Forced network discovery is depicted as the optional step 6 in Figure 7. Forced discovery will result to new discovery messages arriving at System Monitor, after which the process is continued from step 5.

4.3.2 On-Demand Service Injection on Devices

In case that even after forced network discovery of devices no service instances that match the query have been identified, the system tries to inject (i.e. remotely install) appropriate instances of the identified service types (depicted as the optional step 7 in Figure 7). This involves finding devices that are capable of hosting the service, and actually installing it in a platform-dependent way. The new instances are detected by the System Monitor, and again, the process continues from step 5.

Injection is possible if the descriptions of the service types identified in previous steps include installation instructions and executable software artifacts. In the Service Repository, for each service type, we therefore provide data structures for deployable artifacts, including (dynamic and static) hardware requirements and dependency relations between services. These requirements are compared with the capabilities and states of the currently available devices. An efficient service to device mapping is calculated and platform-specific injection actions are taken to change the system according to the mapping. Once the injection finishes successfully, control is handed back to Service Instance Ranking, which uses the Service Monitor again to discover the newly installed Service Instances.

In a concrete example, the service description of a fire detection service could include both DPWS bundle for installation on an DPWS-enabled sensor platform and a rule set for a rule-based sensing system. Meta information makes sure that the bundle and the rule set are only applied to the appropriate platforms. Additional information for deployment can be included, such as the desired coverage (*e.g.* 80 % of all nodes), dependency on other services, *e.g.* a temperature measurement service and a fire shutter control service. If the service to be deployed is depends on other services, those can be deployed as well. Metadata may also include the memory required to install the new service. Our efforts on service mapping and injection are described in detail in [22].

5 Process Evaluation

A prototype of the described process was implemented and integrated to the SOCRADES Integration Architecture. The prototype implementation was developed in Java and deployed on a Java Enterprise Application Server (SAP NetWeaver) at two distinct locations. The evaluation of the prototype was split in three parts following the sub-parts of the Real-World Service Discovery

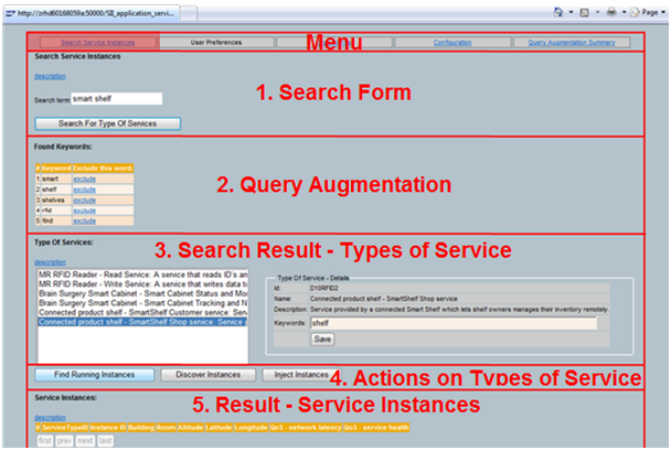

Figure 8. Search User Interface, looking for instances of services in the Application Service Catalogue

Process (i.e. Types Query, Candidate Search, Provisioning).

The first step in evaluating the implementation of the process was to get a number of DPWS-enabled devices offering services to search for. Unfortunately since it is only recently that DPWS has become an official standard (WS-DD) its adoption on industrial devices is ongoing. Thus, we decided to simulate a larger number of devices that one could expect to find in future industrial environments. Since developers usually write the description of Web Services [11], we selected seventeen experienced developers and asked them to write the description of a selected device and of at least two services it could offer. The developers were given the documentation of a concrete device according to the projects they were currently working on. Based on these descriptions we generated thirty types of services (described in WSDL containing DPWS metadata) for sixteen different smart devices ranging from RFID readers to robots and sensor boards. Out of these, one thousand service instances were simulated at the two deployed locations.

This work further extended the simulation by implementing them on a prototyped shop floor in a laboratory and industrial setup in two different scenarios described in section 5.4. The locations spanned across cities and countries using Internet as the communication backbone for these services and internal operations. Common shop floor devices like temperature and vibration sensors were identified. SunSPOT sensors[2], gantry robots, PLC (Programmable Logic Controller) devices controlling conveyor belts and proximity sensors from leading industrial vendors were wrapped with (DPWS) Web services or directly deployed service instances on the devices themselves. We further tested the integration of RESTful devices by implementing a native Web server for the SunSPOTs [18].

The prototype of the SOCRADES Integration Architecture was used in the back end to monitor, search and compose services offered by these devices. The LDU of the SIA was used at different locations in the field trials to dynamically discover devices on the shop floor.

2. www.sunspotworld.com

5.1 Search User Interface

A search (or discovery) Web user interface (UI) for developers was developed (shown on Figure 8) on the top of the Java Enterprise implementation of the RSDDP. This UI offers several interaction zones corresponding to the steps of the process and allows the developer to feed her feedback into the discovery loop at every step of the process.

In the first zone (1. Search Form) the developer is asked to enter n keywords. These keywords are then extended with related words by the Query Augmentation and Assistant module (see Section 4.1.1 and step 1 on Figure 4), as shown in the second zone (2. Query Augmentation). In this zone the developer can also choose to remove/extend words before performing a Service Type Lookup (second part of step 1 and Section 4.1.2). The result of this is shown in the third zone (3. Search Result - Types of Services) that is a list of all the types of services corresponding to the extended keywords. The developer then selects the types she is actually interested in. To help her identifying these, a table provides a description of each type as well as all its currently associated tags. Additionally, the developer can tag the service types with new keywords, if she thinks relevant tags are still missing. Once the service types are selected, the Candidate Search is performed (step 2 on Figure 4 and Section 4.2. The result of this execution (zone 5. Result - Service Instances) is a list of running instances for all the selected types. As already mentioned, to facilitate the final selection of a service, the instances are ranked according to the context parameters extracted on the developer side. As an example the service instances on Figure 8 are sorted according three combined parameters i.e. network latency, service health and location (with respective relevance levels of 10%, 30% and 50%).

Before taking the final decision on what service to select the instances can be tested. In the case of a (DP)WS service a click on a instance results in opening its complete WSDL file, and a further click on a link opens the Web Service Navigator of SAP Netweaver to allow testing of the service. In the case of a RESTful service a click on an instance directly retrieves the identifier of the service (i.e. a URL) and allows for direct testing within the browser.

Finally, if no running instances were found for any of the selected types, the developer can choose to force the network discovery of instances (step 3 on Figure 4 and Section 4.3) or to perform an injection of the selected types on appropriate embedded devices (zone 4. Actions on Types of Service).

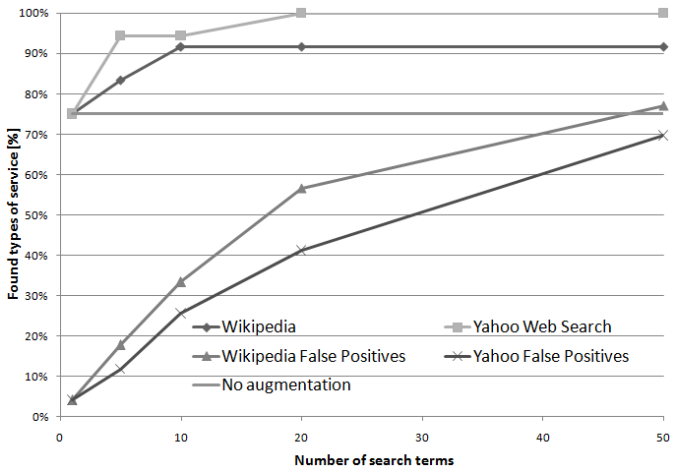

Figure 9. Results for the Query Augmentation with Yahoo! and Wikipedia

5.2 Evaluation of Types Query and Candidate Search

In the evaluation of the Query Augmentation module we wanted to know whether:

1) Augmenting developer input with related keywords could help in finding more real-world Web services
2) What type of combination of query strategies is the most suitable.

Two types of strategies were used. In the first we used a human generated index (i.e. Wikipedia), and in the second a robot generated index (i.e. Yahoo! Web Search). The input keywords were selected by seven volunteers, all working in IT. They provided seventy-one search terms (composed of one to two words) that they would use if they were to search for services provided by the seventeen devices. These terms were entered one by one and all the results were logged.

The trend extracted from these experiments is shown on Figure 9. Two results can be drawn. First the Query Augmentation process does indeed help in finding more real-world services. Without augmentation 75% (plain gray line on Figure 9) of the service types were found and using the query augmentation up to a 100%. However, the Query Augmentation generates a number of false positives, i.e. service types that are returned even if they are not related to the provided keywords (depicted by the two lines at the bottom of Figure 9). Thus we need to restrict the number of keywords added to the intitial ones. The observed optimum is between 5 and 10 added keywords, leading to less than 20% false positives out of 95% types of services found. The second result can be seen on Figure 9 which reveals that using Yahoo!, the approach performs slightly better than when using Wikipedia. Looking more in detail, we see that indeeded, approximately 50% of the keywords used against Wikipedia did not lead to any page, simply because they do not have yet dedicated articles, even if Wikipedia was growing at a rate of about 1400 articles per day[3]. However, when results where extracted from Wikipedia pages they were actually more relevant for the searched real-world services. Thus, a good solution would be to chain the strategies so that first human generated indexes are called and then robot generated ones, in case the first part did not lead to results.

The Candidate Search was evaluated based on a proof of concept implementation. We tested two chained ranking strategies for the generated services; one comparing service health and given weight of 30% as well as one comparing network latency and given a weight of 50 %. They performed as expected, sorting the lists of retrieved service instances according to the ranking strategies which, we believe helps developers finding their way across the results, but would need to be tested with neutral volunteers. We implemented the sorting using the merge sort algorithm which has a complexity of $O(n \log n)$, and since the strategies can be chained we have an overhead for the ranking of $O(mn \log n)$ where m is the number of strategies and n the number of Service Instances.

5.3 Evaluation of On-Demand Service Provisioning

On-Demand Service Provisioning can be the result of not being able to find a suitable running service, or even because the possible existence of such a service on a specific device would result in better satisfying the requirements *e.g.* better performance. Our implementation of the on-demand provisioning part was done using adaptations of well-known algorithms, which resulted to NP-hard approaches for service to device mapping [22]. Both probabilistic/efficient ($O(nk)$) and complete/inefficient ($O(n^k)$) algorithms have been implemented. Some evaluation was done using test scenarios, in which the probabilistic algorithms produced results close to the optimum, with respect to a given objective function. A proof of concept implementation demonstrated the service mapping and deployment both on simulated and real devices (PDA-level). Flexibility is achieved by using exchangeable strategies for each step of the mapping process that can be exchanged at during run-time. This approach is also scalable, since most of the components can be easily replicated and distributed across different locations. A detailed evaluation and discussion of the on-demand service provisioning is given in [22].

5.4 Demonstrator and Field Trial

In the recent years we have witnessed the SOA concepts starting to be successfully applied to the shop floors of future factories. Thus, the key motivation for evaluating the prototype was to identify the feasibility of SOA based cross-location infrastructure in real-world situations such as in factories. In the demonstrated scenario multiple heterogeneous physical devices on the shop floor were

3. http://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia

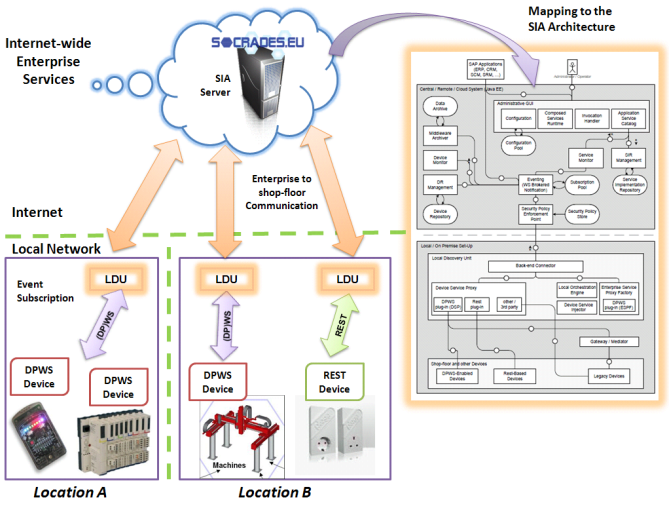

Figure 10. Trial of the SIA in an multi-location, cross-network real-world scenario

discovered, the services to be composed were identified and used based on the RSDPP. In particular, DPWS and RESTful Web services were deployed (see Figure 10) with the aim to evaluate part of the aforementioned concepts on real data and events.

In the trial, a manufacturer operates in two distinct geographical locations (marked as "Location A" and "Location B" in Figure 10), that also belong to different networks and reside behind firewall/proxy without direct connections between the shop-floors. At company's headquarters a running ERP system governs the overall operations with respect to the two locations. A production order is issued from the headquarters to its production facilities at location A. During the course of production, a severe unpredictable failure of machines causes the production to be stopped at location A. Such a scenario occurs often in production plants where the stop per hour could run to a loss of thousands of dollars.

In this scenario the ERP system at the headquarters is immediately alerted about the current loss and an estimated delay in completing the production order is calculated. Subsequently the ERP evaluates alternative scenarios in order to realize the customer's order and satisfy its constraints. Therefore it decides to relocate the remaining of the production order to location B. It also arranges for the already produced parts of location A to be transfered to a storage room where the parts from location B will also arrive shortly, in order to complete one whole shipment to the customer.

During the **design time** of the composite application, the services hosted on the devices at these facilities are discovered by the SIA and its LDUs. These services, along with their QoS and other contextual information, are stored in the Service Type Repository. A sequential process-oriented composite application is created by the developer by searching for instances of services using the RSDPP and the search user interface in order to complete a production sequence. Functionality envisioned at design time but that could not be found at run time is deployed on-demand. This composite application is then also available as a Web service. Two such applications were developed for location A and location B for this scenario. In the second phase - **runtime** phase, the actual composite application is being executed.

6 Conclusion

The future Internet will be highly populated by heterogeneous networked embedded devices that will further blur the borders of real and virtual world, empowering us with new capabilities in creating real-world aware business applications. For this to happen, it is of high importance to be able to find real-world services that can be dynamically included in enterprise applications - a quite challenging task considering the application requirements, technologies and heterogeneity of devices. In that line of thought, we have presented here an approach that would facilitate this task for developers, allowing them not only to search efficiently for services running on embedded devices, but also to deploy missing functionalities on-demand.

The comprehensive process demonstrated in this article shows that we can extend the reach of enterprise computing to the real world (and vice versa). To achieve this, we suggest to use Web Service standards (DPWS) and Web oriented patterns (REST) to easily integrate physical devices into existing enterprise information systems. Web services on devices can be used to dynamically register devices and the service(s) they provide. We have suggested to use queries to search service metadata that has been gathered by the network discovery of devices. Furthermore, we have designed and evaluated automatic augmentation of the search queries, with strategies that extend queries with related keywords found on knowledge repositories available on the network *e.g.* third party Web sites. With this extension we have shown that significantly more services can be identified without overloading devices with description data. We have also shown how context is important for real-world services and explained its use within the service discovery process. Finally, we presented how missing functionality can be injected on devices upon developer's request.

Acknowledgment

The authors would like to thank the European Commission and the partners of the SOCRADES (www.socrades.eu) project for their support.

References

[1] E. Fleisch and F. Mattern, *Das Internet der Dinge: Ubiquitous Computing und RFID in der Praxis:Visionen, Technologien, Anwendungen, Handlungsanleitungen*. Springer-Verlag, 2005.

[2] D. Lizcano, M. Jiménez, J. Soriano, J. M. Cantera, M. Reyes, J. J. Hierro, F. Garijo, and N. Tsouroulas, "Leveraging the upcoming internet of services through an

open user-service front-end framework," in *ServiceWave '08: Proc. of the 1st European Conference on Towards a Service-Based Internet*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 147–158.

[3] P. J. Marrón, S. Karnouskos, and D. Minder, *Research Roadmap on Cooperating Objects*. European Commission, Office for Official Publications of the European Communities, July 2009, no. ISBN: 978-92-79-12046-6.

[4] 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)*, Madrid, Spain, April 2009.

[5] M. Hatler, D. Gurganious, C. Chi, and M. Ritter, "WSN for Smart Industries," OnWorld Study, 2007. [Online]. Available: www.onworld.com

[6] M. Marin-Perianu, N. Meratnia, P. Havinga, L. de Souza, J. Muller, P. Spiess, S. Haller, T. Riedel, C. Decker, and G. Stromberg, "Decentralized enterprise systems: a multiplatform wireless sensor network approach," *Wireless Communications, IEEE*, pp. 57–66, 2007.

[7] W. K. Edwards, "Discovery systems in ubiquitous computing," *IEEE Pervasive Computing*, vol. 5, no. 2, pp. 70–77, 2006.

[8] F. Jammes and H. Smit, "Service-oriented paradigms in industrial automation," *IEEE Transactions on Industrial Informatics*, vol. 1, no. 1, pp. 62–70, Feb. 2005.

[9] L. M. S. de Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio, "SOCRADES: A web service based shop floor integration infrastructure," in *Proc. of the Internet of Things conference (IoT 2008)*. Springer, March 26-28 2008, pp. 50–67.

[10] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao, "Tiny web services: design and implementation of interoperable and evolvable sensor networks," in *Proc. of the 6th ACM conference on Embedded Network Sensor Systems*. Raleigh, NC, USA: ACM, 2008, pp. 253–266.

[11] M. Crasso, A. Zunino, and M. Campo, "Easy web service discovery: A query-by-example approach," *Science of Computer Programming*, vol. 71, no. 2, pp. 144–164, Apr. 2008.

[12] C. Atkinson, P. Bostan, O. Hummel, and D. Stoll, "A practical approach to web service discovery and retrieval," in *Proc. of the International Conferent on Web Services (ICWS 2007)*, 2007, pp. 241–248.

[13] R. Monson-Haefel, *J2EE Web Services: XML SOAP WSDL UDDI WS-I JAX-RPC JAXR SAAJ JAXP*. Addison-Wesley Professional, Oct. 2003.

[14] H. Song, D. Cheng, A. Messer, and S. Kalasapur, "Web service discovery using General-Purpose search engines," in *Web Services, 2007. ICWS 2007. IEEE International Conference on*, 2007, pp. 265–271.

[15] W. Drytkiewicz, I. Radusch, S. Arbanowski, and R. Popescu-Zeletin, "pREST: a REST-based protocol for pervasive systems," in *Proc. of the IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, 2004, pp. 340–348.

[16] D. Guinard, V. Trifa, P. Spiess, B. Dober, and S. Karnouskos, "Discovery and on-demand provisioning of real-world web services," in *Proc. of the 2009 IEEE International Conference on Web Services (ICWS '09)*. Los Angeles, CA, USA: IEEE Computer Society, 2009, pp. 583–590.

[17] E. Wilde, "Putting things to REST," School of Information, UC Berkeley, Tech. Rep. UCB iSchool Report 2007-015, November 2007.

[18] D. Guinard, V. Trifa, T. Pham, and O. Liechti, "Towards physical mashups in the web of things," in *Proc. of INSS 2009 (IEEE Sixth International Conference on Networked Sensing Systems)*, Pittsburgh, USA, June 2009, pp. 196–199.

[19] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Trans. Internet Techn.*, vol. 2, no. 2, pp. 115–150, 2002.

[20] C. Pautasso and E. Wilde, "Why is the web loosely coupled? a Multi-Faceted metric for service design," in *Proc. of the 18th International World Wide Web Conference (WWW 2009)*, Madrid, Spain, Apr. 2009. [Online]. Available: http://www.jopera.org/node/190

[21] M. Hepp, K. Siorpaes, and D. Bachlechner, "Harvesting wiki consensus: Using wikipedia entries as vocabulary for knowledge management," *Internet Computing, IEEE*, vol. 11, no. 5, pp. 54–65, 2007.

[22] T. Frenken, P. Spiess, and J. Anke, "A flexible and extensible architecture for device-level service deployment," in *ServiceWave '08: Proc. of the 1st European Conference on Towards a Service-Based Internet*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 230–241.

[23] S. Karnouskos, O. Baecker, L. M. S. de Souza, and P. Spiess, "Integration of SOA-ready networked embedded devices in enterprise systems via a cross-layered web service infrastructure," in *Proc. ETFA Emerging Technologies & Factory Automation IEEE Conference on*, 25–28 Sept. 2007, pp. 293–300.

[24] P. Spiess, S. Karnouskos, D. Guinard, D. Savio, O. Baecker, L. M. S. d. Souza, and V. Trifa, "Soa-based integration of the internet of things in enterprise services," in *Proc. of the 2009 IEEE International Conference on Web Services (ICWS '09)*. Los Angeles, CA, USA: IEEE Computer Society, 2009, pp. 968–975.

[25] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim, "TinyREST - a protocol for integrating sensor networks into the internet," in *Proc. of the Workshop on Real-World Wireless Sensor Network: SICS*, Stockholm, Sweden, 2005.

[26] C. Pautasso, O. Zimmermann, and F. Leymann, "RESTful Web services vs. "big" Web services: making the right architectural decision," in *Proceeding of the 17th international World Wide Web Conference (WWW 2008)*. Beijing, China: ACM, 2008.

[27] J. I. Vazquez and D. L. de Ipiña, "mRDP: An HTTP-based lightweight semantic discovery protocol," *Computer Networks Journal - Special Issue on Innovations in Web Communications Infrastructure*, vol. 51, no. 16, 2007.

[28] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, March 1995.

[29] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[30] W. Balke and M. Wagner, "Through different eyes: assessing multiple conceptual views for querying web services," in *Proc. of the 13th international World Wide Web conference*. New York, NY, USA: ACM, 2004, pp. 196–205.

[31] A. Schmidt, M. Beigl, and H. Gellersen, "There is more to context than location," in *Proc. of the International Workshop on Interactive Applications of Mobile Computing (IMC98)*, vol. 23, Nov. 1998, pp. 893–901.

[32] R. Weizheng, "A rapid acquisition algorithm of WSN-aided GPS location," in *Proc. Second International Symposium on Intelligent Information Technology and Security Informatics IITSI '09*, 23–25 Jan. 2009, pp. 42–46.

[33] A. Segev and E. Toch, "Context-based matching and ranking of web services for composition," *IEEE Transactions on Services Computing*, vol. 99, no. 1, pp. 210–222, 5555.

[34] H. Chan, T. Chieu, and T. Kwok, "Autonomic ranking and selection of web services by using single value decomposition technique," in *Proc. IEEE International Conference on Web Services ICWS '08*, 23–26 Sept. 2008, pp. 661–666.

[35] V. X. Tran and H. Tsuji, "QoS based ranking for web

services: Fuzzy approaches," in *Proc. 4th International Conference on Next Generation Web Services Practices NWESP '08*, 20–22 Oct. 2008, pp. 77–82.

[36] S. Karnouskos and A. Izmaylova, "Simulation of web service enabled smart meters in an event-based infrastructure," in *Proc. 7th IEEE International Conference on Industrial Informatics INDIN 2009, Cardiff, UK*, June 23–26, 2009, pp. 125–130.