Discovery and On-Demand Provisioning of Real-World Web Services

Dominique Guinard[1,3], Vlad Trifa[1,3], Patrik Spiess[2], Bettina Dober[1], and Stamatis Karnouskos[2]

[1]SAP Research Zurich, Switzerland
[2]SAP Research CEC Karlsruhe, Germany
[3]Institute for Pervasive Computing, ETH Zurich, Switzerland
Email: dominique.guinard@sap.com

Abstract

The increasing usage of smart embedded devices is blurring the line between the virtual and real worlds. This creates new opportunities for applications to better integrate the real-world, providing services that are more diverse, highly dynamic and efficient. Service Oriented Architecture is on the verge of extending its applicability from the standard, corporate IT domain to the real-world devices. In such infrastructures, composed of a large number of resource-limited devices, the discovery of services and on demand provisioning of missing functionality is a challenge. This work proposes a process, its architecture and an implementation that enables developers and process designers to dynamically discover, use, and create running instances of real-world services in composite applications.

1 Introduction and Related Work

The Internet of Things envisions vast numbers of embedded devices, such as networks of sensors and actuators, industrial production lines and machines, and household appliances that are being interconnected and possibly collaborate to provide advanced services [6]. The functionality and sensor data these devices will be offering, are often referred as *real-world services* because they are provided by embedded devices, which are part of the physical world. Unlike most traditional enterprise services, which are virtual entities, real-world services provide real-time data about the physical world.

A study from OnWorld [9] projects that wireless sensor network (WSN) systems and services will be worth $ 6.6 billion in 2011, and in 2012 it is expected that 25.1 million WSN units will be sold for smart home solutions only, a significant increase from the 2 million in 2007. The business opportunities for real-world services are great. As mass market penetration of networked embedded devices is realized, services taking advantage of the devices' novel functionality will give birth to new innovative applications and provide both revenue generating and cost saving business advantages. From a technological point of view, the key challenge is how to discover, assess, and efficiently integrate the new data points into business applications.

Several efforts have explored the integration of real-world and enterprise services e.g. [12, 5]. However, the protocols used do not offer uniform interfaces across the application space and are thus complicated 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 Services standards (SOAP, WSDL, etc.) directly on devices [11, 4, 14]. Implementing WS-* standards on devices presents several advantages in terms of end-to-end integration and programmability by reducing the needs for gateways and translations between the components. This enables the direct orchestration of services running on devices, with high-level enterprise services, e.g. offered by an Enterprise Resource Planning (ERP) applications. For example, if sensors physically attached to shipments could offer Web Services, those could be easily integrated in a process that updates the status (e.g. temperature) and location of the shipment directly in the involved ERP systems.

However, SOA standards were designed primarily for connecting complex and rather static enterprise services. As a consequence, implementing WS-* standards directly on devices is not 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. Additionally, real-world services are found in highly dynamic environments where devices and their underlying services degrade, vanish, and possibly re-appear. As such, this infrastructure can not be considered static and long-lived, as traditional enterprise services. This implies the need for automated, immediate discovery of devices and services as well as their dynamic management.

A crucial **challenge** for SOA developers and process designers is to find adequate services for solving a particular task [3], which for enterprise services implies a manual query of a number of registries, typically UDDI (Universal Description and Discovery and Integration) registries. The results depend largely on the quality of the data within the registry, which is entered manually therefore is prone to error. While this de facto approach is adequate for a rarely changing set of large-scale services, the same does not hold 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 service discovery, different lines of research have been undertaken in order to provide alternatives or extensions of the UDDI standard [3, 15, 1]. However these do not take into account the particular requirements of real-world services.

Based on our previous experience within the SOCRADES project [4], we introduce here a set of requirements to facilitate the use of real-world services within enterprise applications:

1. **Minimal Registration Effort**. A device should be able to advertise its services to a registry using network-level 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.

2. **Support for Dynamic Search**. It should be possible to use also external resources to better formulate the queries. Furthermore, queries should take into account context (e.g. location, Quality of Service (QoS), application context). Support for context is essential as the functionality of most devices is task-specific within well-defined places (e.g. a building, a plant, etc.).

3. **Support for On-Demand Provisioning**. Services on embedded devices offer rather atomic operations, such as obtaining data from sensors. Thus, while the WSN platforms are rather heterogeneous, the services that the sensor nodes can offer share significant similarities and could be deployed on-demand per user request.

The key **contribution** of the work presented here is a **service discovery process for real-world services** shown on Figure 1 and detailed in Section 3. The goal of this process, called Real-World Service Discovery and Provisioning Process (RSDPP), is to **assist the developers in the discovery of real-world services** to be included in composite

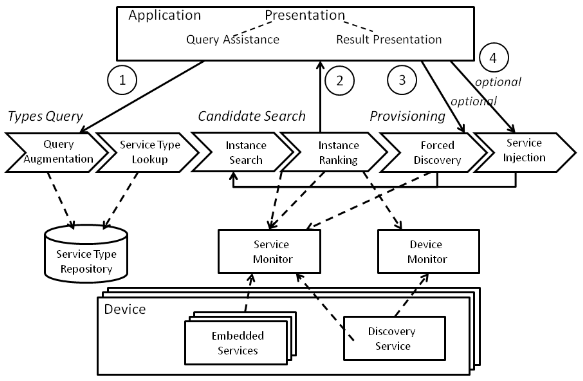

Figure 1. Overview of the complete Real-World Service Discovery and Provisioning Process (RSDPP).

applications. This innovative process fulfills the three requirements of real-world services we described above:

The minimal registration effort requirement is met by using the Device Profile for Web Services (DPWS) [2, 11] and its discovery mechanism. DPWS defines a limited set of WS-* standards which are implementable on resource-constrained devices. A DPWS compliant device also has a set of built-in services, fulfilling the automatic network discovery of devices and services on a local network. We will describe DPWS in Section 2.

We further ensure the minimal registration effort and support for dynamic search by extending user provided keywords with vocabularies of related terms also known as "lightweight ontologies" [10]. 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 3.1.

The dynamic search requirement is also fulfilled by taking into account the user context and matching it with the extracted context of real-world services. This information is then used when retrieving and ranking services as explained in Section 3.2. The requirement for on-demand dynamic provisioning is fulfilled by a software architecture that enables the developer to automatically deploy services on devices when no satisfying service was found in the environment [7]. This architecture is described in Section 3.3.

Finally, we present our implementation within an enterprise application (based on Java Enterprise Edition and SAP NetWeaver) to validate our results and show its usability for actual deployments in Section 4.

Before describing the process itself we start with an overview of the framework in which the RSDPP was developed and 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 within the research project SOCRADES, and has been implemented as part of the SOCRADES Integration Architecture (SIA) [4]. The role of SIA is to enable the integration of real-world services running on embedded devices within enterprise-level applications. Web Services standards constitute the standard 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 using a high-level, abstract interface based on Web Services. This allows 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 details of the underlying hardware.

The SIA has been described in [4], and here we only describe a few particular components of the whole architecture. In SIA, the lowest layer is called the Devices Layer and comprises the different embedded devices that are running the different services. SIA is able to interact with devices using several communication protocols, such as DPWS, OPC-UA, etc. In this article, however, we focus solely on devices that are connected to SIA using web services (DPWS). Nevertheless, 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. Furthermore, SIA allows applications to subscribe to any events sent by the devices, offering a publish/subscribe component by providing a WS-Notification compliant Notification Broker. 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.

On top of the Device Layer, we have built higher-level components to ease the management and use of devices in a standardized and uniform way. The **Device Repository** holds all known static device information (metadata) of all on-line and off-line devices, 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.

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 1. 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.

Device Profile for Web Services (DPWS) The Devices Profile for Web Services (DPWS) defines a minimal set of implementation constraints to enable (secure) Web Service messaging, discovery, description, and eventing on resource-constrained devices. Its objectives are similar to those of Universal Plug and Play (UPnP) but, in addition, DPWS is fully aligned with Web Services technology and includes extension points, allowing for seamless integration of device-provided services in enterprise-wide application scenarios. The DPWS specification defines an architecture in which devices run two types of services: a) hosting services and b) hosted services. *Hosting services* are directly associated to a device, and play an important part in the device discovery process. *Hosted services* are mostly functional and depend on their hosting device for discovery. Hosting services comprise of: **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.

Network Discovery of Devices As real-world services run natively on embedded devices, we need a robust mechanism to dynamically find devices as they connect to the network, and dynamically retrieve metadata about the device and the services it hosts. Furthermore, we want mechanism to fulfill the requirement for a minimal registration effort. To achieve this, we use DPWS which follows the WS-Discovery specification. When a new device joins the network, it will multicast a `HELLO` message via the UDP protocol. By listening to this 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 SOCRADES Device Monitor, containing the device's static metadata. The metadata information can be classified into a certain set of metadata *classes*, 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.

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 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 1 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 he would search for documents on any search engine. This query is then extended with related keywords fetched from different websites, and used to retrieve types of services that describe a 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 1, 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 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).

3.1 Types Query

In the first part of the discovery process (step 1 on Figure 1), the developer or process designer enters keywords describing the type of service she wants to find (step 1 on Figure 2). 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 will be sent to the Query Augmentation module which is going to extend the query with additional keywords. The output of this module is then used to retrieve and rank types of services.

3.1.1 Query Augmentation and Assistant

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

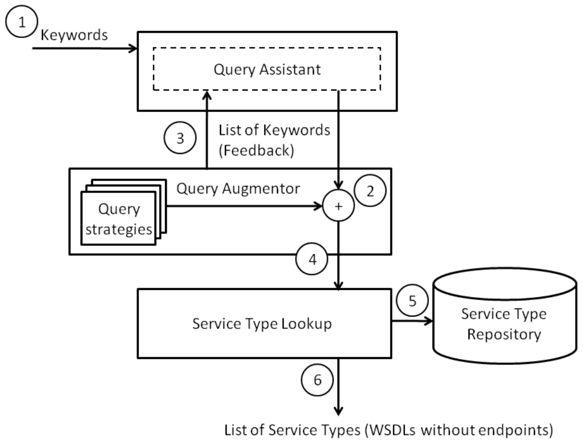

Figure 2. Looking for a Service Type.

mechanism is that it lacks flexibility. For instance, a developer wants to find services offered by a "smart meter", a term often used to describe a device that can measure the energy consumption of other devices. 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" keywords. We want to avoid the construction of domain ontologies, and to minimize the amount of data embedded devices need to provide upon network discovery of device 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 2.

The idea is to use existing knowledge databases, such as web encyclopedias (e.g. Wikipedia) and search engines (e.g. Google, Yahoo Web Search), in order to extract "lightweight ontologies" [10] or vocabularies of terms from their semi-structured results. The basic concept of the query augmentation (step 2 on Figure 2) is to call $1..n$ web search engines or encyclopedias with the search terms provided by the user, 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 keywords "smart meter" and can hence 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* and their structure is based on the Strategy Pattern [8], which enables 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. 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 2).

In order to test the query augmentation mechanism we implemented a number of Query Strategies, they are evaluated in Section 4.1.

3.1.2 Service Type Lookup

The augmented query is used to determine any matching service types in the Service Type Repository (step 4 and 5 on Figure 2). 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, which was sent to the Service Monitor upon network discovery or extended by manual entry. This includes human readable descriptions, contact information, legal terms, explicit keywords 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.

3.2 Candidate Search

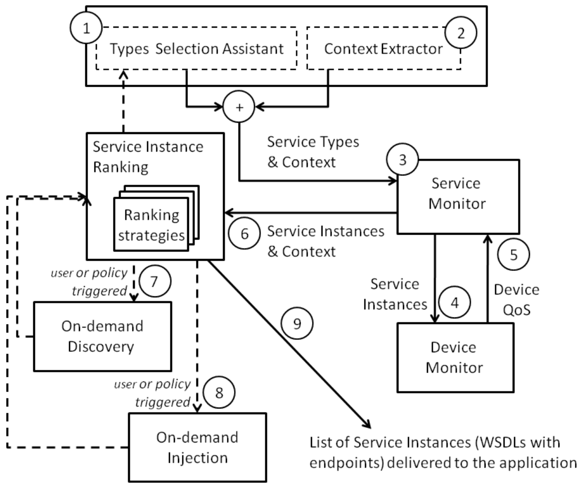

Figure 3. Ranking and optionally Provisioning Service Instances.

Real-world devices are volatile e.g. connect and disconnect thus the need to decouple the discovery of service types from the discovery of actual instances of services. The Candidate Search (step 2 on Figure 1) 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 3). Alternatively she can select all the types retrieved in the Types Query part of the process.

3.2.1 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. Context is information that qualifies the physical world, and 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. Context extraction on the developer side is done at step 2 of Figure 3. It is worth noting that context on the developer side reflects her expectations, this information is then going to be compared to the service and device side context by the Service Instance Ranking component (see Section 3.2.3).

To fit our requirements, the context is modeled by two distinct parts: 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 in.

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 scope of the discovery. The **QoS Information** reflects the expectation of the developer (or of the application she is currently using) in terms of how should the discovered service 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 (JSR 179). The **Address** encapsulates the virtual description of the current location, with information such as building name, building 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.

3.2.2 Service Instances Search

In step 3 of Figure 3, the identifiers of the selected service types and the full context object 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 2) the Service Monitor and the Device Monitor know what devices are currently providing what service types. In steps 4 and 5 of Figure 3, the Service Monitor queries the Device Monitor for the quality of service of the selected service instances. This information is derived from polling the devices from time to time as well as by monitoring the invocations of services and calculating their execution time.

The QoS information is then packed into a context object that contains all available contextual information for each device running the selected Service Instances. Since we can not expect every device to provide a full contextual profile, the Service Monitor has its own default context component which can be used to extend the information the device provides.

3.2.3 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 his machine. As shown in step 6 of Figure 3 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 example, a Ranking Strategy could use the network latency so that the services will be listed sorted according to their network latency.

Weighted and chained combinations of Ranking Strategy can be used when sorting. Furthermore, 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 3. Thus, instances can be ranked against each other or/and 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 user.

3.3 On-Demand Service Provisioning

In case no running service instance has been found, On-Demand Service Provisioning will be carried on. This will first dynamically discover devices on the network that offer services matching the requirements of the developers. In the last instance, installation of services on suitable devices will be carried out.

3.3.1 Forced Network Discovery of Devices

According to our experience, network discovery of devices is not entirely reliable. This is because it uses UDP and might take a long time to propagate across the whole system. Sometimes fresh information is needed and thus, another discovery mode is proposed that is particularly suited for environments where devices with unknown capabilities continuously connect to or disconnect from the network. Forced network discovery of devices is responsible for dynamic discovery of specific devices. The dynamic process can use different types of filters that specify the type of the device or a scope in which the device resides and/or other semantic information. This is useful to restrict the result set when looking for new devices, as only devices that match the criteria specified will respond. Filter information is included in a *Probe* message sent to a multicast group; devices that match the probe 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 *Resolve-Match* 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, available sensors).

3.3.2 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 found, the system tries to generate appropriate instances by injecting (i.e. remotely installing) the identified service types. This involves finding devices that are capable of hosting the service, and actually installing them in a platform-dependent way. This 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 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 finished successfully, control is handed back to Service Instance Ranking, which will use 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. Further information for deployment can be included, like 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 annotated to depend on other services, those can be deployed as well. Further metadata may include the flash or RAM memory required to install the new service. Service mapping and injection is described in greater details in [7].

4 Process Evaluation

A prototype of the described process was implemented and integrated into the SOCRADES Integration Architecture. The prototype implementation was developed in Java and deployed on a Java Enterprise Application Server (SAP NetWeaver) in two different locations.

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 DPWS is a rather new standard, its adoption on industrial devices is still ongoing. Thus, we decided to simulate a number of devices that one could expect finding in industrial environments. Since developers usually write the description of Web Services themselves [3], we selected 17 project-neutral 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 their own skills and professional background. Based on these descriptions we generated 30 types of services (described in WSDL containing DPWS metadata) for 16 different smart devices ranging from RFID readers to robots and sensor boards. Out of these, 1000 service instances were simulated at the two deployed locations. A proof of concept evaluation of the whole process was done using this prototype implementation and the simulated data.

4.1 Evaluation of Types Query

In the evaluation of the Query Augmentation module we wanted to know whether: 1) Augmenting user input with related keywords could help in finding more real-world Web Services, and 2) What type of query strategies is the most suitable. Two different types of Query Strategies have been compared. In the first type, we used a human generated index (Wikipedia), and in the second a robot generated index (Yahoo Web Search). The input keywords were selected by 7 volunteers, all working in the IT domain. They provided 71 search terms (composed of one or two words) that they could imagine using when searching for services provided by the 17 devices. These terms where entered one by one and all the results were logged.

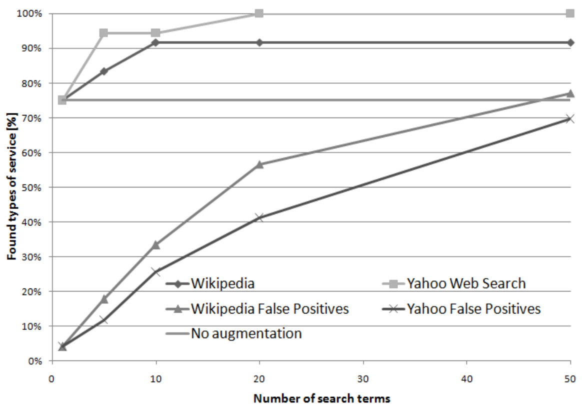

Figure 4. Results of the Query Augmentation with Yahoo and Wikipedia.

The trend extracted from these experiments is shown on Figure 4. Two results can be identified: First, the Query Augmentation process does help in finding more real-world services. Without augmentation 75% of the service types were found, using the query augmentation up to a 100%. However, 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. Thus we identified the need for restricting the number of keywords added to the initial ones. The observed optimum was between 5 and 10 added keywords, resulting in 95% percent of types of services found for less than 20% false positives. Secondly, although Figure 4 suggests that using Yahoo seems to perform slightly better than Wikipedia, that result needs to be explained. The main reason is that about 50% of the keywords used against Wikipedia did not lead to a page because they did not have dedicated articles yet. However in those cases, where results were actually extracted from Wikipedia pages, they appeared more relevant to searched real-world services. Furthermore Wikipedia grows at a rate of about 1500 articles per day[1]. Thus, a good solution would be to chain the strategies so that first human generated indexes are called and then, as a fallback, robot generated ones in case the first part did not lead to results.

4.2 Evaluation of On-Demand Service Provisioning

Our implementation of the on-demand provisioning part was done using adaptations of well-known algorithms. The service to device mapping has been proven to be NP hard [7]. Both probabilistic / efficient ($O(n \cdot k)$) 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 opti-

[1]`http://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia`

mum, in regards to a given objective function. A proof of concept implementation showed the service mapping and deployment both on simulated and real, PDA-scale devices. Flexibility is achieved by using exchangeable strategies for each step of the mapping process that can be exchanged at run-time through configuration. Besides, this approach is 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 [7].

5 Conclusion and Future Work

In a future highly populated by networked embedded devices, finding real-world services that can be dynamically included in enterprise applications will be a challenging task. In that view, 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 when needed. The comprehensive process demonstrated in this article shows that we can extend the reach of enterprise computing to the real world. To achieve this, we suggest to use Web Services standards to easily integrate physical devices into existing enterprise information systems. Web services on devices (in particular DPWS) can be used to dynamically register devices and the service(s) they provide. We have suggested to use queries to search services 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 databases available on third party web sites. With this extension we have shown that significantly more services can be found without overloading devices with description data. We also show how context is important for real-world services and explain its use within the service discovery process. Finally, we presented how missing functionalities can be injected on devices upon developers' request.

Future work will include a thorough evaluation of the whole process and its architecture, focusing on performance, scalability and usability. Finally, we are deploying the architecture in real-world trials in order to better understand the uses and limitations of the approach.

Acknowledgments

The authors would like to thank the European Commission and the partners of the European IST FP6 project *Service-Oriented Cross-layer infRAstructure for Distributed smart Embedded devices* SOCRADES (`www.socrades.eu`), for their support. Thanks also to Dr. Patrik Fuhrer (University of Fribourg, Switzerland) for his help on the software engineering part of the process.

References

[1] 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)*, pages 241–248, 2007.

[2] H. Bohn, A. Bobek, and F. Golatowski. SIRENA - service infrastructure for real-time embedded networked devices: A service oriented framework for different domains. In *Proc. of the International Conference on Networking, Systems, Mobile Communications and Learning Technologies*, page 43. IEEE Computer Society, 2006.

[3] M. Crasso, A. Zunino, and M. Campo. Easy web service discovery: A query-by-example approach. *Science of Computer Programming*, 71(2):144–164, Apr. 2008.

[4] L. M. S. de Souza, P. Spiess, D. Guinard, M. Koehler, S. Karnouskos, and D. Savio. Socrades: A web service based shop floor integration infrastructure. In *Proc. of the Internet of Things (IOT 2008)*. Springer, 2008.

[5] W. K. Edwards. Discovery systems in ubiquitous computing. *IEEE Pervasive Computing*, 5(2):7077, 2006.

[6] E. Fleisch and F. Mattern. *Das Internet der Dinge*. Springer, 1 edition, July 2005.

[7] T. Frenken, P. Spiess, and J. Anke. *A Flexible and Extensible Architecture for Device-Level Service Deployment*, volume 5377 of *LNCS*, pages 230–241. Springer, December 2008.

[8] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Nov. 1994.

[9] M. Hatler, D. Gurganious, C. Chi, and M. Ritter. WSN for Smart Industries. OnWorld Study, 2007.

[10] M. Hepp, K. Siorpaes, and D. Bachlechner. Harvesting wiki consensus: Using wikipedia entries as vocabulary for knowledge management. *Internet Computing, IEEE*, 11(5):54–65, 2007.

[11] F. Jammes, A. Mensch, and H. Smit. Service-oriented device comunications using the devices profile for web services. In *Proc. of 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC05) at the 6th International Middleware Conference*, 2005.

[12] 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*, 2007.

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

[14] 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*, pages 253–266, Raleigh, NC, USA, 2008. ACM.

[15] 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*, pages 265–271, 2007.