Semantic Middleware for the Internet of Things

Zhexuan Song, Alvaro A. Cárdenas, Ryusuke Masuoka
Fujitsu Laboratories of America, Inc.
{zhexuan.song, alvaro.cardenas-mora, ryusuke.masuoka}@us.fujitsu.com

Abstract

The Internet of Things (IoT) refers to extending the Internet to devices such as home appliances, consumer electronics, and sensor networks. As multiple heterogeneous devices attempt to create area networks, one of the major challenges is the interoperability and composability of their services. The traditional way to address interoperability is to define standards; however, there are many standards and specifications that are incompatible with each other. In this paper we propose an application layer solution for interoperability. The key idea is to utilize device semantics provided by existing specifications and dynamically wrap them in our middleware into semantic services. Next, with the help of Semantic Web technologies, users can create and then execute complex tasks involving multiple heterogeneous devices. We demonstrate how our framework automates interoperability without any modifications to existing standards, devices, or technologies, while providing to the user an intuitive semantic interface with services that can be executed by combining devices in the network.

1   Introduction

The Internet of Things is expected to revolutionize the way we interact with devices at work, home, or public spaces. By providing network connectivity to embedded devices, everyday objects will be able to interact with each other, introducing new services and opportunities to end users.

For example, a typical Home Area Network (HAN) can connect a set of devices such as computers, printers, streaming clients, and set top boxes. The diversity of devices connected to these networks is expected to increase rapidly in the next couple of years due to the global interest in smart energy devices such as refrigerators, thermostats, and smart meters.

As new devices enter the market, the proliferation of heterogeneous technologies pose serious challenges for interactions among devices following different specifications.

To address service interoperability concerns, several industry alliances and standard organizations have defined interoperability standards such as Bluetooth [6], UPnP [27], DLNA [11], SLP [12], Zeroconf [26] (Apple's implementation of Zeroconf is called Bonjour [7]), and Infrared [13].

Each standard plays an important role in facilitating interoperability so that devices can successfully communicate with each other as long as they comply with the standard. However, a standard alone can not solve the interoperability problem where multiple standards exist. For example, if a user takes a picture with a mobile phone that supports Bluetooth, she will have no trouble in printing the image on a printer that also supports Bluetooth. However, if there is a UPnP-based media player in the HAN, she may have trouble displaying the picture on the media player. Unfortunately, we argue that there will never be a shortage of standards, mainly because "Things" are diverse in many aspects.

In addition to having multiple incompatible standards, there is another problem with the current practice of standards-based interoperability approaches: standards need to reach specific agreements on the interaction of their services a priori; in other words, specifications have to anticipate all possible future scenarios [21]. This makes it difficult to dynamically connect an ever-growing (and changing) set of devices and services in HANs.

A possible solution to these problems is to extend standards with adapters to other standards. The drawbacks of this idea are two-fold. First, one standard has no direct control over other standards. Therefore, whenever something changes in another standard, the adapter has to be revised according to the modification. Secondly, in order to support interoperability among N standards, $O(N^2)$ adapters have to be created, which is clearly inefficient. Another possible solution is to introduce bridges among standards–e.g., a device that can translate among different specifications; however, it is not clear if this is a general approach and if it can be used for composing multiple services.

To address some of these concerns, we implemented a middleware based on semantic web technologies for improving the automatic configuration of an heterogeneous network. Semantic Web approaches can provide

(1) interoperability between devices and information, (2) context awareness to applications, thus reducing the search space for service discovery and composition, and (3) meaningful information to users so they can decide how to compose multiple services and improve security and privacy decisions (semantic data can be better understood by "things" as well as humans compared to current protocol descriptions).

In particular, we propose the use of a semantic layer where each device is mapped to a semantic service, which we call the "Service Layer." A service is associated with at least one semantic description in OWL-S [20]. Services are the abstraction of devices and the service layer shields users from the complexity of directly dealing with devices of different standards and makes it easy for users to compose services for accomplishing complex tasks.

There are several benefits in this approach. First, this solution is not a "single ultimate standard," in the sense that we are not requesting any additional work from the device manufacturers: our support of various standards is completely transparent. Its implementation is more efficient: we map different standards to the same abstraction in the service layer; therefore, in order to achieve the interoperability among N standards, only $O(N)$ adapters have to be created. We also provide an intuitive semantic interface so the user can understand the multiple services that can be provided by the network. Finally, our framework is easy to extend because the amount of work required to support an additional standard is limited.

We have implemented our semantic-web framework to show the interoperability and composability of services of devices following non-IP Bluetooth and UPnP specifications.

2 Background and Related Works

2.1 Service Discovery and Device Interoperability

Connecting a set of heterogeneous devices in a single network can create interoperability challenges. Simply connecting two devices together via Ethernet or WiFi, allows them to communicate using the IP protocol; however IP communication does not guarantee that their services will be compatible. The most basic way to address this problem is to install drivers. For example, if you connect a printer and a computer to a network, you may need to install the driver of the printer in the computer.

This traditional solution will not scale if you connect multiple heterogeneous devices to the network who may want to access the printer. Not only is installing a driver for each device burdensome, but drivers may not even be available for embedded devices ("things") connected to the network.

Service-discovery protocols were introduced to address these challenges. If two devices implement a given service discovery protocol, they will be able to find the services provided by each other and use these services.

Bluetooth, for example, provides a service discovery protocol (SDP) which allows devices to discover services supported by other devices and their associated parameters. When connecting two Bluetooth devices, SDP is used for determining the *Bluetooth profiles* (e.g., video distribution profile, hands-free profile, or the upcoming smart energy profile). Two products implementing Bluetooth technology leverage standardized profiles. If a cellphone and a printer are Bluetooth-compatible, the printer will be discovered and configured without additional drivers or software.

There are several other service discovery solutions such as UPnP [27], Zeroconf [26], Jini [14], DLNA [11], Infrared [13], and SLP [12]. While all these solutions facilitate interoperability, a user will still have to understand which solution their products implement and only purchase compatible devices.

2.2 Semantic Web and OWL-S

The semantic web has a different vision of the Internet of Things. It is a vision of information that is understandable by "things," so that machines can perform more work involved in finding, combining, and acting upon information on the web. These tasks are well suited for enabling the interoperability of different resources on the web. In fact, the semantic web has been positioned as a way to achieve automated interoperability among diverse and complex web services.

Semantic web services are key components of the semantic web. Their goal is to automate the discovery and orchestration of services on the web. Semantic web services provide semantic annotations to regular services to facilitate a higher degree of automation on how machines can interpret, discover, and compose different services.

OWL-S [20] is an Ontology Web Language (OWL) used for describing semantic web services. An OWL-S semantic service description consists of three main parts: *profile, process,* and *grounding.*

The profile defines what the services do. It includes the general information of a service (name, description, semantic input/output) and it allows users to manipulate the service (such as service composition) in the semantic layer.

The process defines the input, outputs and parameters. If a service involves multiple groundings, the work flow (how the groundings are put together, what are the input/output mappings between groundings, and so on) is also defined in the process part.

The grounding part enables users to invoke services. It includes the details that are required during the interaction with the service (entry point and parameters).

In addition to describing semantic web services, OWL-S also enables service composition, where services can be combined in order to perform an aggregated task. Composition consists of two steps: synthesis (selecting services that will participate in creating the composed service), and orchestration (execution of the composed service).

2.3 Related Work

Developing new middleware solutions for pervasive computing is an active area of research [23]. The main services that a middleware solution must provide are context management [2], service configuration mechanisms [17], and interoperability.

Conventional middleware technologies such as CORBA, and Java-RMI achieve interoperability by standardizing a common set of protocols and formats in order to interact with other devices. In this paper we are interested in allowing middleware to negotiate the protocols used to interact with others at runtime.

An example of middleware in this class is Jini [14]. In Jini, each Java object can be a service, and the information about the services is maintained by a central lookup server. There are many extensions for interoperability in Jini. For example, Allard et al. [1] introduced a framework to combine Jini with UPnP, and Kasper and Buhrer [15] addressed how to discover Bluetooth devices in Jini.

A similar technology (also based in Java) was proposed by Rellermeyer et. al. [22]. In this architecture, each device is loosely coupled as a software module in R-OSGi (a specification for distributed design and management of Java software modules). Their approach is independent of the network protocol, and their extensions support services in Bluetooth with other service discovery protocols based on TCP/IP. Their architecture does not incorporate the semantic information of the services and therefore it does not provide the same flexibility that the semantic web offers for the composition of services.

Web services are another popular approach for providing interoperability in embedded devices. The Service Oriented Architecture (SOA) is a promising technology for integrating devices into a business IT network. By running web services on smart objects, SOA can create an Internet of services the facilitate the interoperability and availability of these devices with back end applications such as Enterprise Resource Planning (ERP) systems. SODA [8] is a promising approach for integrating SOA principles into the Internet of Things. Similarly, the European project SOCRADES [9] provides a middleware layer so that web service-enabled devices can connect to enterprise applications such as ERP systems.

The problems that SOA, SODA and SOCRATES address are similar to the the ones the semantic web tries to solve: the integration of many heterogeneous devices through web services. The semantic web can build on top of these services to enhance collaboration with a formal description of concepts, terms and relationships between devices. These opportunities have been recognized; for example, one of the fundamental requirements for the SOCRADES project is the support of semantic web concepts [9], and as such, our approach can be integrated as part of their architecture.

The usefulness of the semantic Web in ubiquitous computing has been recognized in several scenarios. Lassila articulated how the semantic web has the potential to solve the interoperability nightmare introduced by ubiquitous computing [18]. The Perci project [5, 24] uses semantic web services for describing the services provided by physical objects. Their goal is the composition of multiple services for mobile devices interacting with the physical world–they do not work on interoperability. The Ubiware project [16] is another framework for using semantic web technologies in the Internet of things; however, they have not studied the interoperability between different service discovery protocols.

Non-middleware solutions for interoperability have also been proposed in the literature. For example, approaches for allowing interoperability between Bluetooth and UPnP include the work of Ayyagari [3], who extended the Bluetooth discovery profile to support UPnP, and Delphinato et. al. [10] who proposed a proxy that presents Bluetooth services as a UPnP embedded device containing one or more services. As we pointed out in Section 1, it is not easy to develop a general solution from these ad hoc efforts efforts.

3 Interoperability at Semantic Layer

We apply semantic web technologies to device services in order to enable users to focus on the tasks they want to achieve, rather than how to achieve them. To this end we have created a framework called Task Computing [25, 19]. In our framework, semantics fulfill two roles: (a) it enables the manipulation of and the interaction with the computing environment at the semantic layer so that users are able to define and execute complex tasks without having to worry about the details of underlying devices and services. and (b) it is a medium for interoperability between the underlying disparate resources. Figure 1 illustrates the architecture of the Task Computing framework.

In this paper we introduce our extensions to the Task Computing framework by focusing on interoperability at the semantic layer. We follow three steps: first, middleware discovers various devices and generates the corresponding semantic services. Next, a semantic user interface helps users to build tasks as service compositions. Finally, the task is completed by executing individual devices.
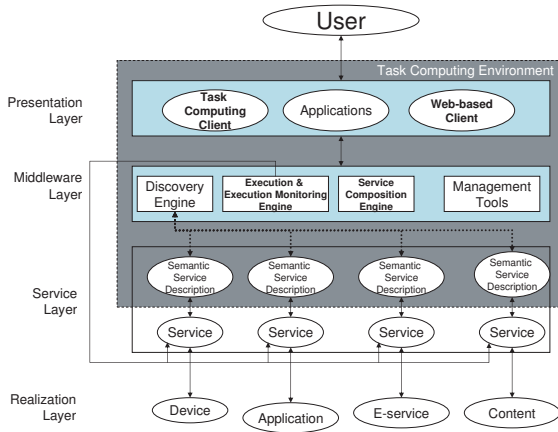
Figure 1. General architecture of Task Computing Environment. Task Computing provides a common platform for interoperability among devices, application, e-services and contents. Modules that are related to this work include the discovery engine and the execution engine at the "Middleware Layer." For each standard, the corresponding discovery module wraps devices into semantically described services at the "Service Layer." The Task Computing Environment (marked in a gray box) could be running in the device that a user can directly control (such as her smart phone).

3.1 Semanticizing Devices

Our first step is to create dynamically a semantic service description through the standard that the device follows, and to represent the device as a semantically described service.

We create a new discovery module for each standard (see Figure 2). The discovery module for a standard works as follows. First, the module uses the discovery protocol defined in the standard to locate the devices. After a device is detected, the information of the device is extracted from the data obtained through the protocol and an OWL-S semantic service description is created internally. We are not creating the description from scratch. Instead we maintain a "semantic service description template repository" and store a set of OWL-S templates for each known device type from the standard. The template file is a valid OWL-S description except some fields which will be instantiated based on the information we retrieved from the device. The last job of the discovery module is to fill these fields and register the "complete" semantic service description. We call this procedure "semanticization."

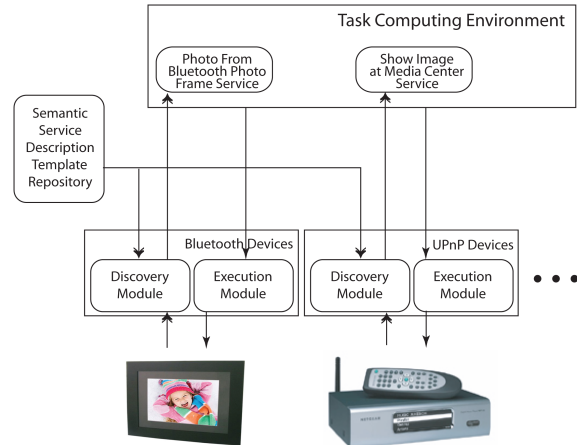As we mentioned in section 2.2, the grounding part

Figure 2. Device semanticization and execution: A Bluetooth enabled digital phone frame and a UPnP based media center are semantically described as services in Task Computing and allow the user to display the image from the digital phone frame on the media center.

of a valid OWL-S semantic service description should consist of all information that is required to execute the service. In theory, once the values of the input parameters of the service are known, the service can be properly invoked just based on the grounding part. In OWL-S, unfortunately, only WSDL grounding is formally defined and there is no support for other standards. Therefore, we extend the OWL-S grounding definition by introducing new grounding support for device-specific information. Due to the space limitation, we do not list these details here.

3.2 Building Tasks

During the second step, Task Computing framework will help a user to build tasks out of available services. Some of the functionalities provided are: (1) Based on the semantic input and output types of the services, the framework will check the validity of possible compositions. (2) When some parts of the task are determined, services that can not be further added into the task will be eliminated, thus limiting the search space. (3) As a user adds a new service into the task, the framework will find appropriate places to put the service within the task. (4) The framework allows a user to input keywords, then suggests the possible tasks, and sorts the tasks based on the preferences of the user.

3.3 Device Grounding

After a user constructs a task, the last step is to execute it. A task is a workflow of services. (Figure

4 shows a *task* example.) During the task execution, each service will execute in the order defined in the workflow. When the execution of one service is done, based on the flow, the output is transferred to other services.

There are three steps involved in executing a service: (1) marshalling service inputs to parameters, (2) invoking the corresponding realization (device, web service and so on), and (3) unmarshalling parameters to service outputs. Among them, step one and three are optional.

The execution engine of Task Computing supports multiple execution modules, such as the WSDL based web service execution module, the REST based web service execution module, and others. We also provided additional execution modules for different standards, which will be introduced in section 4.3.

4 Implementation

Our solution leverages semantics readily available explicitly and implicitly in each standard. To date, none of the standards adopt semantic web technologies for its device description, but nonetheless, device semantics are abundant in each technology. How the semantics are given differs from one technology to another and their semantics are not always given in a machine-processable way.

While our approach is general and can enable interoperability of different standards, in this paper we use as an example Bluetooth and UPnP service discovery and configuration protocols. (Note that Bluetooth has a TCP/IP profile that supports IPv6; however not all devices in the market implement the IP stack. In addition, even if the IP stack was available among all Bluetooth devices, manufacturers would still need to select among one of the many IP service discovery protocols such as UPnP, Zeroconf, SLP, etc.)

4.1 Semantics in Devices

4.1.1 Semantics in Bluetooth

The basic unit in the Bluetooth standard is a *Device*, and each device may have one or more *Bluetooth Services*. Each Bluetooth device contains a device name and a 24-bit code called *Class of Device* (or CoD). *CoD* contains information about the type of the device, and the type of the available Bluetooth services on that device. The 24-bit code is divided into *Major Service Classes* (11 bits), *Major Device Class* (5 bits), *Minor Device Class* (6 bits), and 2 spare bits which are reserved for the *CoD* format type.

Figure 3 shows the fields of the 24-bit *CoD*. Starting from the most significant bit, the first 11 bits (bit number 23 to bit number 13) represent the *Major Service Classes*. *Major Service Classes* is a high level generic
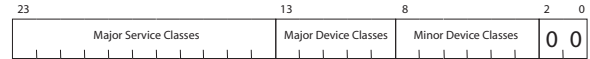

Figure 3. 24-bit Bluetooth Class of Device (CoD) fields.

category of service class, which tells us what types of services the device will provide. For example, if the 18th bit is set to one, this means that the device provides a *Rendering* service (Printing, Speaker, ...), and if the 20th bit is set to one, this means that the device provides an *Object Transfer* service (v-Inbox, v-Folder). The classification is very coarse, and we rarely use it to determine the device type.

The next 5 bits represent the *Major Device Class*, which define a general family of the device. For example, if the value of *Major Device Class* is 00010, 00100, 00110, the device belongs to *Phone*, *Audio/Video*, *Imaging* class. Under the *Major Device Class*, the device can be further classified into *Minor Device Class*, which is represented by the following 6 bits of the *CoD*. For example, under *Imaging* major class, there are four *Minor Device classes*, which are *display* (00XXX1), *Camera* (00XX1X), *Scanner* (00X1XX) and *Printer* (001XXX). Here X means "don't care." A complete mapping table is available as part of the Bluetooth standard. *CoD* is available when a Bluetooth device is discovered. As we check the major and minor device classes, we are able to determine what kind of device it is.

Next we further query the Bluetooth device to retrieve the *Service Record*. *Service Record* consists entirely of a list of Bluetooth service attributes. The key attribute is the *Service Class*. The Bluetooth service is an instance of a *Service Class*, which provides an initial indicator of the capabilities of the service, and defines what other attributes, including their types and semantics, must or can appear in the service record. *Service Classes* are specified using Universally Unique Identifier (UUID) numbers, values for which are predefined.

4.1.2 Semantics in UPnP

In the general UPnP architecture, a basic unit is a *Device*. *Device* contains zero or more *UPnP Services*. Under each *UPnP Service*, a set of *Actions* is defined. XML descriptions are provided at both *Device* and *UPnP Service* level. The *Device* description includes the general information about the device (name, manufacturer, version, etc.), the list of *UPnP Services*, the general information about each *UPnP Service*, and the URL where the description of each *UPnP Service* is located.

Under the general UPnP architecture, there are a set of sub-standards defined for various types of devices, such as *MediaServer*, which provides media

content; *MediaRenderer*, which consumes media content; and *Control Point*, which controls the media flow. Under the UPnP Device XML Description, the type is defined under "deviceType" tag. For example, Media Service device has type "urn:schemas-upnp-org:device:MediaServer:1." By tracking this field, we can figure out what the device is designed for.

The *UPnP Service* description consists of the details of *Actions* which are the basic execution unit in the UPnP standard. Similarly, according to the standard, the types of *UPnP Services* are also fixed, which is defined under "serviceType" tag. For example, a *MediaServer* may consist of a *ContentDirectory*, whose service type is "urn:schemas-upnp-org:service:ContentDirectory:1;" a *ConnectionManager*, whose service type is "urn:schemas-upnp-org:service:ConnectionManager:1;" and optionally an *AVTransport*, whose service type is "urn:schemas-upnp-org:service:AVTransport:1." After checking the service type, we will be able to retrieve the semantics of each *UPnP Service*.

The lowest level of UPnP standard is the *Action*. Standards are provided for actions as well. For example, under *UPnP Service ConnectionManager*, there are three required actions: "GetProtocolInfo," "GetCurrentConnnectionIDs" and "GetCurrentConnectionInfo," and two optional actions: "PrepareForConnection," and "ConnectionComplete." The meaning of each action is clearly described in the standard, which means that we can easily use a semantic service description in OWL-S to describe the actions and wrap them as semantically described services.

4.2 Semanticization

In this section we show how to semanticize a Bluetooth picture printer. Other devices and standards (e.g. UPnP) follow a similar procedure.

Our Bluetooth discovery module uses *BlueCove* [4] programming API, which is a LGPL (Lesser General Public License) licensed JSR-82[1] implementation developed by Intel Research. As the Bluetooth discovery module detects a new device, it will first ask for the Bluetooth CoD (*class of device*). From the CoD, the module finds out that the *major device type* is "Imaging" and *minor device type* is "Printer." At this point, we know that this device is designed to print images.

Next, from the template repository, the discovery module will pick a "Print Image" template. The "Print Image" template is describing a semantic service that takes a "ImageFile" as the input and has no output. ("ImageFile" is a class defined in an associated ontology that is widely used in the Task Computing project.) The template has open fields, including the service name, the service description, and the Bluetooth service record. The module will query the device

---

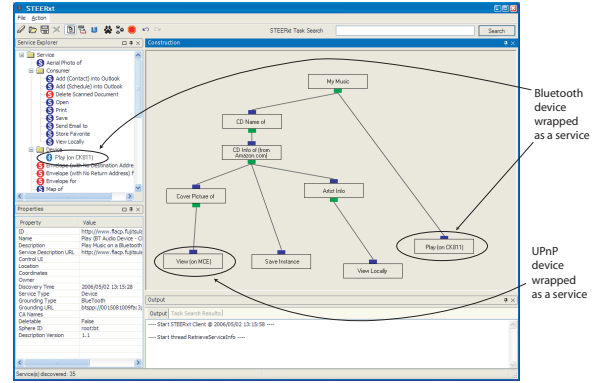[1] JSR-82 is the official Java Bluetooth API.

Figure 4. Screenshot of a Task Computing client. In this task, a user can select a MP3 song from her music repository, play it on the Bluetooth speaker CK811, get the CD information about the song, show the cover picture of the CD and view information about the artist. This specific task involves Bluetooth and UPnP devices, Web Service provided by Amazon, and OS functions.

(using BlueCove API), extract the information, and fill these fields. When the semantic service description is "complete," the description is added into the Task Computing service layer. Please note that Task Computing requires that each service have a unique service ID, which is also listed in the semantic service description. The creation of the unique ID is also part of the Task Computing discovery module's job.

Once the semantic service description is generated and added into the Task Computing service layer, the device is treated as just another semantic service. From the upper layer's point of view (Figure 1), there is no difference between this and any other existing services. Now users have complete freedom to build tasks out of them.

4.3 Device Grounding

Generally speaking, the first step is to prepare the parameters for the invocation of the corresponding realization, since the values of the service inputs do not always match the required parameters. According to the OWL-S specification, it is possible to use an XSLT script within the "xsltTransformationString" tag to define the transformation.

Although the XSLT script is adequate in some cases, as devices are involved, there are many situations when an advanced mapping is required and the mapping cannot be defined by a script in "xsltTransformationString." Let us use a Bluetooth based "Print Image" as an example. The input of the service is an instance

of "ImageFile." From this instance, we can either get the URL of the image, or the image file itself encoded in Base64. In either case, we can not directly use it as the parameter and start to invoke the Bluetooth device. Instead, we have to download the image first (or decode the image), then check which printing protocol the Bluetooth device supports. The device may either support the *OBEX pushing*, the *Basic Image Profile*, or other similar protocols. For a specific printing protocol, such as the *OBEX pushing*, we will further follow the OBEX communication procedures. Additional steps may include: requesting a connection, splitting the image if it is too big, closing the connection after finishing the push, and so on. Clearly, these steps can not be handled only by an XSLT script. Our solution for the above limitation of the XSLT script is to introduce additional execution modules for different standards. During the service execution, the Task Computing execution engine will determine which execution module it should use based on the semantic service description of the service. Then the engine will hand over the inputs to the corresponding modules along with the grounding part of the semantic service description. The grounding part is important because it includes the information that is required for the invocation. All the details are handled within each execution module.

After the job is done, the Bluetooth execution module will collect the outputs if any, send them back to the Task Computing execution engine. The engine will continue the task.

In this solution, we isolate the execution behaviors specific to each standard inside each execution module, thus it is easy to extend the solution and support new standards in the future.

4.4 Trial System Deployment

We deployed a Task Computing system in our conference room. The conference contains devices like projector, TV, phone, printers, and many others. Devices implement different specifications. For example, CK811 speaker is a Bluetooth based speaker system, MCE is a UPnP based center. Under the same environment, we also deployed some software based services and web based services, and a task computing middleware. When the middleware is launched, it discovered the devices using different specifications first. On finding a new device, the middleware wraps the device with a semantic description. Meanwhile, the task computing middleware finds the semantic description of other services.

After the initial discovery stage, the conference environment is ready to use. Task computing framework provides several choices of clients. Our framework provides several choices for clients. We used a PC based client to discover and utilize devices and services. When a visitor enters the conference room, she uses the client to talk with the task computing middleware, and retrieves available services. Please note that devices are wrapped as services, and the difference between device and other service is transparent to the visitor. From the client, the visitor can build tasks (with the help of the client) and execute them.

Figure 4 shows a screenshot of a task computing client. The left top corner is the discovery pane. A Bluetooth device "ORA Wireless Speaker CK811" is wrapped and exposed as the semantic service "Play (on CK811)." Along with a UPnP device "Play (on MCE)" and other services, an interesting task is composed. By executing the task, a user can select an MP3 song from her music repository (folder), and the CD name is extracted. By invoking the Amazon web service for search, the information of the CD is retrieved. Then the CD cover image will be shown on the UPnP media center, the information of the artist will be given, and the MP3 song will be played on the Bluetooth enabled CK811 speaker.

5 Discussion

Semantics brings benefits in guiding service composition to end users. When services are wrapped with semantic descriptions (OWL-S descriptions), the system filters out unreasonable matches between services and provides better suggestions for tasks (as service compositions). Therefore, even an ordinary user with very little knowledge about services available in an environment can easily achieve her own tasks without being overwhelmed by a large number of services.

Although the benefits of using Semantic Web technologies have been recognized, their adoption has been slow. One main reason is the lack of familiarity with "semantics" by the community at large, and the natural resistance to learning a new language. This is partly due to the complexity of semantic web languages. Even if device manufacturers invest in learning and deploying semantic services, the costs of might be significant. In this work we address these problems by capturing and utilizing readily available semantics from existing technologies and standards. Our solution does not require device manufacturers to produce new semantic descriptions for their devices. Instead, from the protocols already used by those devices, the Task Computing framework internally generates semantic descriptions on-the-fly. We believe that this approach will bring down the current difficulty in adoption of semantic web technologies and pave the way for wider adoption.

In our realization, we are using a centralized control mechanism: namely all protocol translations are done within the Task Computing environment (although we can have multiple Task Computing devices working distributively in our network). Compared with a distributed approach in which we wrap individual devices with a protocol translation module, the centralized so-

lution requires no changes on the existing devices. It is important because many devices have limited capabilities, and they can not handle complex requirements. In addition, a centralized approach is more cost-efficient, as we can reuse the radios and specific communication modules from the central translator.

In our deployment, we use a template library which contains semantic service description templates maps to some known types of devices. This allows us to quickly semanticize devices on-the-fly. However, if the system encounters an unknown device type, the system will not find a template and therefore, cannot accept the new device. Moreover, when the number of templates increases, there will be an increase in management costs, and it may impact the scalability of the system.

6 Conclusion and Future Work

In this paper, we introduced a comprehensive solution for interoperability among heterogeneous devices in the context of the Internet of Things. Our solution relies heavily on semantic web technologies, but we use them internally to our system. It extracts semantics available from the existing standards the device uses and describes them semantically on-the-fly. With dynamically and internally provided semantic descriptions, the devices are treated as semantic services and users can enjoy the benefits brought by semantic web technology when they build and execute tasks. The solution hides complicated device details from users and does not require any changes to the existing devices. The same idea is applicable to a large number of existing web resources that have not yet been semantically described.

References

[1] J. Allard, V. Chinta, S. Gundala, and G. Richard. Jini meets UPnP: an architecture for Jini/UPnP interoperability. In *Proceedings of Application and the Internet*, 2003.

[2] S. Arbanowski, P. Ballon, K. David, O. Droegehorn, H. Eertink, W. Kellerer, H. van Kranenburg, K. Raatikainen, and R. Popescu-Zeletin. I-centric communications: Personalization, ambient awareness, and adaptability for future mobile services. *IEEE Communications Magazine*, pages 63–69, 2004.

[3] A. Ayyagari. Bluetooth esdp for upnp, 2001. http://www.comms.scitech.susx.ac.uk/fft/bluetooth/-ESDP_UPnP_0_95a.pdf.

[4] Blue cove. http://sourceforge.net/projects/bluecove/.

[5] G. Broll, E. Rukzio, M. Paolucci, M. Wagner, A. Schmidt, and H. Hussmann. Perci: Pervasive service interaction with the internet of things. *IEEE Internet Computing*, 13(6):74–81, 2009.

[6] Bluetooth technology. https://www.bluetooth.org.

[7] Bonjour. http://www.apple.com/support/bonjour/.

[8] S. de Deugd, R. Carroll, K. Kelly, B. Millett, and J. Ricker. SODA: Service oriented device architecture. In *IEEE Pervasive Computing*, volume 5, 2006.

[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 *The Internet of Things*, Lecture Notes in Computer Science, pages 50–67. Springer Berlin/Heidelberg, 2008.

[10] A. Delphinanto, J. J. Lukkien, A. M. J. Koonen, F. T. H. den Hartog, A. J. P. S. Madureira, I. G. M. M. Niemegeers, and F. Selgert. Architecture of a bi-directional bluetooth-upnp proxy. In *Consumer Communications and Networking Conference, 2007. CCNC 2007. 4th IEEE*, pages 34 –38, jan. 2007.

[11] Digital living network alliance. http://www.dlna.org.

[12] E. Guttman. Service location protocol: Automatic discovery of IP network services. *IEEE Internet Computing*, 3(4), 1999.

[13] Infrared data association. http://www.irda.org.

[14] Jini. http://www.jini.org/.

[15] S. Kasper and L. Buhrer. Jini discovers bluetooth, 2002. http://www.tik.ee.ethz.ch/ beutel/projects/-sada/2002ss_sa_vincent_bt_jini.pdf.

[16] A. Katasonov, O. Kaykova, O. Khriyenko, S. Niktin, and V. Terziyan. Smart semantic middleware for the internet of things. In *5th International Conference on Informatics in Control, Automation and Robotics*, pages 169–178, May 2008.

[17] J. King, R. Bose, H.-I. Y. a nd Steven Pickles, and A. Helal. Atlas: A service-oriented sensor platform: Hardware and middleware to enable programmable pervasive spaces. In *Proceedings of the 31st IEEE Conference on Local Computer Networks*, pages 630–638, 2006.

[18] O. Lassila. Applying semantic web in mobile and ubiquitous computing: Will policy-awareness help. In *Semantic Web and Policy Workshop, 4th International Semantic Web Conference*, 2005.

[19] R. Masuoka, Y. Labrou, and Z. Song. Semantic web and ubiquitous computing – task computing. *AIS SIGSEMIS Bulletin*, 2004(3):21–24, October 2004.

[20] OWL-S: Semantic markup for web services. http://www.daml.org/services/owl-s/1.0/owl-s.html.

[21] T. Payne and O. Lassila. Semantic web services. *IEEE Intelligent Systems*, 19:14–15, 2004.

[22] J. S. Rellermeyer, M. Duller, K. Gilmer, D. Maragkos, D. Papageorgiou, and G. Alonso. The software fabric for the internet of things. In *The Internet of Things*, Lecture Notes in Computer Science, pages 87–104. Springer-Verlag, Berlin Heidelberg, 2008.

[23] G. Schiele, M. Handte, and C. Becker. *Handbook of Intelligence and Smart Environments*, chapter Pervasive Computing Middleware, pages 201–227. Springer, 2009.

[24] S. Siorpaes, G. Broll, M. Paolucci, E. Rukzio, J. Hamard, M. Wagener, and A. Schmidt. Mobile interaction with the internet of things. In *Embedded Interaction Research Group*, 2004.

[25] Z. Song, Y. Labrou, and R. Masuoka. Dynamic service discovery and management in task computing. In *1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, 2004.

[26] D. Steinberg and S. Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., 2005.

[27] UPnP forum. http://www.upnp.org.