Semantic Service Discovery in a Middleware Based Ubiquitous Environment

Md. Kamrul Hasan, Lenin Mehedy, Sungyoung Lee, Young-Koo Lee

***Abstract*— A roaming user in ubiquitous environment should have access to different services anywhere anytime. In an infrastructure based smart environment, user acquires this facility from the middleware. Syntax based service discovery has proven to be inadequate for flexible interaction between user and the middleware; context based semantic matching is necessary. This paper shows the use of ontology to facilitate the semantic discovery of services in a ubiquitous middleware named Context Aware Middleware for Ubiquitous System (CAMUS).**

I. Introduction

With the increase of processing power, mobile devices are now hosting more resource intensive applications [1]. Current trend is to make the handheld a personal assistant, to make it the controller of all the devices. As an example, a person carrying a PDA will like his device to control the office, set up a meeting, or to print a document.

To make intelligent decisions, the PDA needs to know its surroundings or the context. However, the PDA may not have all the equipments to infer all the contexts it needs. Moreover, it does not have the model of the domain or the semantic meaning of the world that is very necessary to infer high level context from low level sensor data. A ubiquitous middleware can provide the required contexts through context services to the application residing in the PDA. Middleware also provides flexible discovery of device services so that the application can control the environment through those services. Towards that end, we built Context Aware Middleware for Ubiquitous System (CAMUS) [2], [3].

CAMUS is a distributed middleware built on jini's [4] Service Oriented Architecture (SOA). However, CAMUS is not the only context aware middleware; there are other middlewares out there. To name a few: GAIA [5], AMUN [6], SOCAM [7] which are built on CORBA [8], JXTA [9] and OSGI [10] respectively. Our focus is not to show contrast with other middlewares rather to show the specialty of the service delivery mechanism of our middleware.

In ubiquitous environment, an easy and flexible mechanism is needed to deliver the services to the applications or users. User interaction steps should be reduced for mobile devices. The service discovery framework and the application should negotiate between themselves without explicit interference of the user. The negotiation can be enabled through semantic annotation of the services. Though lot of researches are going on to add semantics to web services [11], [12], [13], little effort has been made to enable semantic discovery of middleware services in ubiquitous environment. Presumably, the reason is that applications are built specifically for a middleware in a single domain; for example middleware for smart home, smart office etc. On the contrary we model CAMUS for multiple domains such as: Home, Office, Super Markets, where users will carry their PDAs with smart applications running on them. In this multi domain environment, a user may need to rediscover the services or find alternative services as he moves to another domain. Currently, none of the well-known middlewares such as CORBA, JINI, JXTA, OSGI (see [14] for survey) and the ubiquitous middlewares built on them can provide semantic service discovery. They can only provide attribute, value pair based matching, which is pretty rudimentary for our requirements. So, we put our efforts to add semantics to the CAMUS services.

There are several unique features of our work. First we used OWL-S ontology for describing the middleware services. This enables us to convert or wrap any of the CAMUS services to web service. We included dynamic attributes for device services and some Quality of Context (QoC) attributes like freshness and precision for context services. Lastly, the use of NASA space ontology makes CAMUS domain ontology standardized and interoperable.

The rest of the paper is organized as follows. In section II, we describe scenarios that motivate our research. In the subsequent two sections we describe the architecture of CAMUS and service discovery and registration mechanism based on jini. Section V and section VI describes Service ontology and query processing. We contrast our work with existing approaches in section VII. We conclude with a conclusion & future works section.

II. Scenario

Suppose the user was in his office and was viewing an important document. As he comes to corridor, his document is shown on a near by display. Then he drives his private car to the market. He could not finish reading the document and so after reaching the market, he asks his personal assistant in the PDA to show the document again. It then looks for a display but finds that all of them are occupied. So, from the middleware in the market it gets a number of alternative suggestions. The first suggestion can be to print the document from the near by printer. A few minutes later, the user remembers that he did not turn off the devices in his office. So

Manuscript received July 01, 2006. This research was supported by the MIC (Ministry of Information and Communication), Korea under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment) in collaboration with SunMoon University. Professor Young-Koo Lee is the corresponding author.

The authors are with Real-Time & MultiMedia Lab, Department of Computer Engineering, Kyung Hee University, 449-701, Republic of Korea. Their emails are {kamrul, lenin,sylee}@oslab.khu.ac.kr, yklee@@khu.ac.kr respectively.

he starts the room control application, which connects to the middleware in his office building via web, gets the status of the devices in his room and adjusts them accordingly.

Here we will point out a number of issues regarding service discovery. First, we assume the user was being tracked by an RFID tag attached to the PDA inside his office room. The whole office building cannot be furnished with RFID tags. So, when he comes out of his office room, he is tracked by a WLAN tracking system. The quality of these services varies. As we can assume the RFID system is more precise than the WLAN tracking, the context services need to be tagged with quality of service attributes like precision and freshness, so that the application chooses the context services accordingly. Second, some devices have dynamic attributes, like the display can be occupied or the printer may be out of toner, the load of the printer may be high. These attributes should be updated as the values change. Third, the discovery mechanism should not stop even if there is no device of a given service type. It should look into the service type hierarchy and find a closest type of service as an alternative. Fourth, as the user moves out of his office domain, the services may not be accessible through the jini mechanism; we may need to access them as web services. The use of semantic web service description mechanism or service ontology can help us warp a jini service to a web service as needed without much effort. Fifth, the market may not be a CAMUS domain; it can be a middleware from other providers. Even in that case, the personal assistant should be able to interact with the middleware. To restrict the scope we will not discuss the fourth and fifth point in this paper.

III. Architecture

CAMUS (see Fig. 1) is a context aware middleware based on jini. Currently CAMUS hosts two types of services, context services and device services. Context services uses CAMUS infrastructure for their operations like location service,

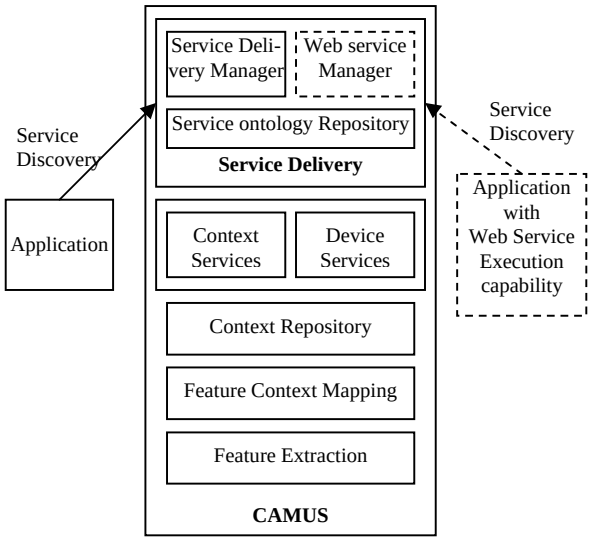

Fig. 1. Architecture of service discovery mechanism on CAMUS

activity recognition service etc. The services receive quantized features from *Feature Extraction* layer and map the features into context in the *Context Mapping* layer. The contexts are then put into the *Context Repository*. Contexts are made following the CAMUS domain ontology which is actually the space ontology from NASA [15]. Context services are built to provide these contexts to the applications. Device services, on the other hand, are actuators or controllers to the devices. The context services, applications and the device services form a close loop and keep the environment controlled.

The top layer of CAMUS is the *Service Delivery*. It is the interfacing module with the application. It has a sub module named *Service Delivery Manager* that discovers and delivers the services to the application. Service Delivery Manager has a lookup and registration mechanism that is specialized for jini clients. We are going to describe that in the next section. *Service Delivery Manager* uses the ontology stored in the *Service Ontology Repository* for its discovery mechanism. The dotted component *Web Service Manager* and the corresponding *Application* are not yet developed and left as our future work.

IV. Service Discovery and Registration Mechanism

Services advertise themselves with attributes and the services are needed to be discovered and delivered to the applications with maximum flexibility. Services should provide service name, service type, service location, vendor, and any other attributes that can help identifying the service. For example, in case of a printer service, the attributes can be service type, service name, service location, resolution, load etc. When an application wants to locate a service, it provides a set of query parameters. These parameters are compared with the available services' attributes and the matched service is returned to the application.

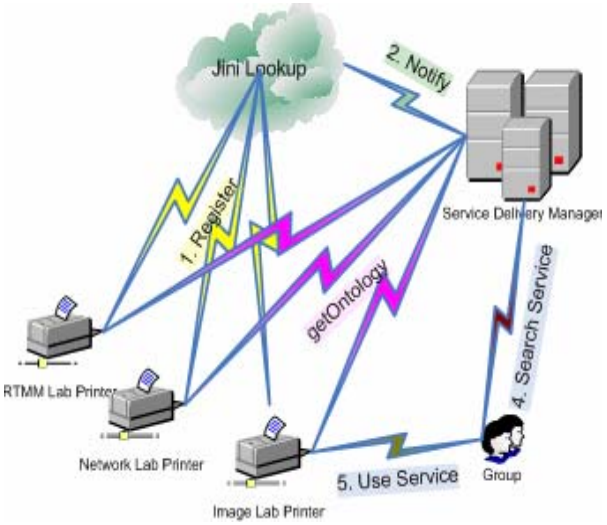

Fig. 2. Registration and discovery mechanism of CAMUS services

In CAMUS, we developed the discovery mechanism without modifying JINI lookup service. We expose *Service Delivery Manager* as a JINI service. When a service registers with the lookup, a notification is sent to *Service Delivery Manager*. It then asks the service to provide its ontology string by calling the remote method `getOntology()`. It then associates the service id with the ontology and puts the ontology in the *Service Ontology Repository*. When the service expires, lookup sends a notification to the *Service Delivery Manager*. *Service Delivery Manager* gets the service id, and removes the ontology from its database.

V. Service Ontology

The purpose of ontology is standardization. So, we should use well defined semantics to describe our services. We modeled our context and devices as services following the concept of web service and we annotated those services using OWL-S [16].

OWL-S is a language to describe web services semantically. It supplies a set of markups for describing the properties and capabilities of web services in unambiguous, computer-interpretable form. Services are represented by three things: profile, process and grounding.

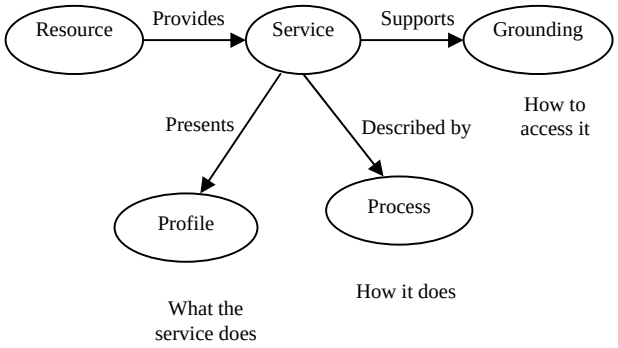

Fig. 3. A high level view of OWL-S ontology

Profile describes the service. It provides information needed by the discovery manager. Process model describes the working procedure of the service. Service grounding specifies communication protocol, message formats, and other service-specific details such as port numbers used in contacting the service.

Generally speaking, the service profile provides the information needed for an agent to discover a service. As our current system does not use automated service execution, we did not use service process and service grounding. OWL-S profile was enough to serve our purpose.

A. Service Hierarchy

In a ubiquitous environment, there can be thousands of services. One very natural technique to manage them effectively or discover them with flexibility is to organize them in categories. Service hierarchy naturally incorporates subsumption relationship among services.

We implement the service hierarchy using OWL profile hierarchy [16]. OWL-S profile does not define any fixed hierarchy of services, it is left open to market trend. So, for our purpose, we define all the middleware services as *CamusService*s. *CamusService* has two subcategories: *ContextService* and *DeviceService*. *ContextService*s are mainly context aggregation services that provide graceful delivery of context to the application. Currently, we have only three such services: *LocationAggregator*, *ActivityAggregator* and *EnvironmentalAggregator*. *DeviceService*s can be *InputDeviceService* or *OutputDeviceService*. Fig. 4 shows the profile hierarchy for CAMUS services.

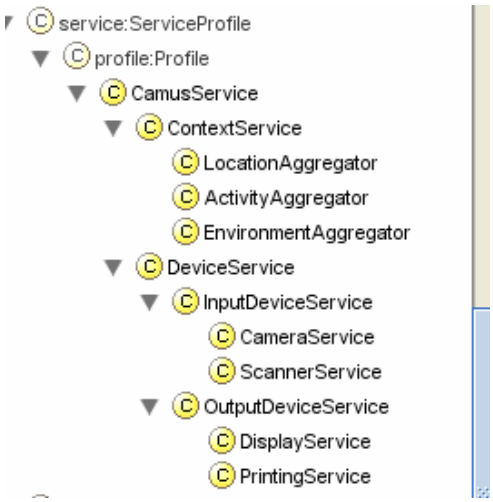

Fig. 4. A view of CAMUS service classes generated by Protégé [17]

B. Service Attributes

B.1. Defining static and dynamic Attributes

Services may have static or dynamic attributes. For example, load of a printer is dynamic property that needs to be retrieved from the service periodically. We consider both types of attributes in our implementation. We used OWL-S `<profile:Parameter>` and `<process:Parameter>` to define the static attributes of services. However, OWL-S does not provide any mechanism to state dynamic attributes in services. So we extended the `<process:Parameter>` class as `<camus:DynamicParameter>` and added an additional property `hasUpdateInterval` in that.

When a service registers with the *Service Delivery Manager*, the *Service Delivery Manager* gets the ontology from the service, and looks for any `DynamicParameter` tag in it. If it gets any Dynamic parameter, it starts a daemon thread that fetches the value from the service periodically as specified in the property `<camus:hasUpdateInterval>`. The ontology snippet below shows the ontology for a printer having the dynamic attribute `hasLoad`:

```
<PrintingService rdf:ID="RTMM_Printer">
                          .
                          .
<profile:hasParameter>
<camus:DynamicParameter rdf:ID="hasLoad">
<process:parameterValue
rdf:parseType="Literal">0</process:parameterValue>
<camus:hasUpdateInterval>60
</camus:hasUpdateInterval>
<process:parameterType rdf:datatype=
"http://www.w3.org/2001/XMLSchema#anyURI">
Integer</process:parameterType>
</process:DynamicParameterParameter>
</profile:hasParameter>
</PrintingService>
```

The daemon threads uses polling mechanism to fetch the dynamic attributes which may create network congestion and overhead on the middleware. So, we employed a separate interrupt mechanism also. Service delivery manager provides an interface method `updateAttribute()`. In this case, the service is responsible to decide when to update the dynamic attribute value. If the service updates the attribute value manually, the timer of that dynamic attribute in the service delivery manager is shifted by the time specified in `hasUpdateInterval`.

The addition of new ontology makes our implementation domain specific. We argue that, as `<camus:DynamicParameter>` is inherited from `<process:Parameter>`, applications from semantic web domain (if CAMUS services are opened as semantic web services) can still consider it as `<process:Parameter>` and continue their operation.

B.2. Quality of Context (QoC) attributes

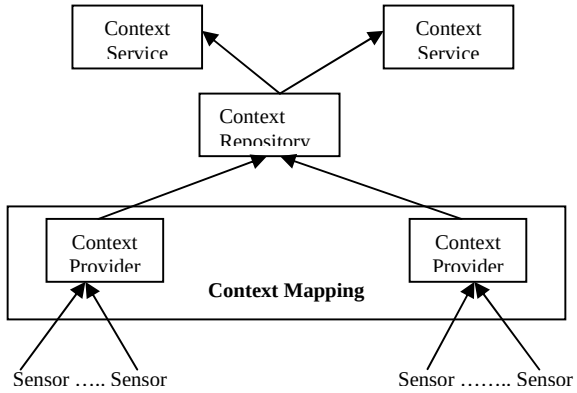

Fig. 5. Layers of context provisioning in CAMUS

Contexts come from different sources. A location context provider can be based on RFID tags or WLAN signal strength. Context Providers reside in the *Context Mapping* layer and continuously formulate contexts and store them in the context repository (see Fig.5). Context Services based on these contexts should provide some hints to the application about the quality of the contexts they are providing. So, we include two QoC attributes in Context Services namely, precision and freshness. Precision is the accuracy of the service where as freshness denotes the update interval of the context. Rather than using exact units to measure them, we use a scale of ten. The higher the value, the better the precision or freshness.

VI. Query Processing

In CAMUS, services can be retrieved based on service category or service attributes. CAMUS can also provide any alternative service if the default matching fails. However, we do not switch between ContextServices and DeviceServices. Currently the system provides three kinds of operators: *comparison operators (Equals To, Greater Than, Less Than, Not Equal), Like operator* and *Near operator*. *Equal To* operators, if used exclusively, give the *EXACT* matching as jini provides. However, other *comparison operators* and *Like operator* give flexible service matching. *Service category* attributes and *Near* operators are evaluated semantically using the OWL-S ontology and the existing CAMUS ontology. *Near* operators provide context based semantic search utilizing CAMUS. We also provide mechanism to relax some of the query attributes if the default query fails.

Application provides a query array of 4-tuples for matching, where each of the 4-tuples consists of attribute *name, operator, attribute value* and *relaxation (yes/no).* An example query is like this:

SerivceCategory	Equals To	PrintingService	Yes
hasName	Like	RTMM	No
hasSpeed	Greater Than	10	Yes
hasLoad	Less Than	30	Yes
hasLocation	Near	Prof	No

We use Jena [18] to browse through the service ontology and retrieve the appropriate service. We generate an RDQL [18] query from the array of 4-tuples excluding the tuples containing *Near* operator. If we get an empty result set, we make the RDQL again, discarding the 4-tuples that have relaxation equal to “yes”.

Then we shortlist the result set based on the Near operator. If the resulting service set is empty after the short list, we reevaluate the services based on only the 4-tuples with Near operator having relaxation No. If the final service set is still empty and serviceCategory attribute is relaxed in the query array, we try to find an alternative service. For this, we find a sibling serviceCategory in the hierarchy and run the query again with the query array. In this case, we discard those attributes that are not present in the sibling serviceCategory.

A. Near operator evaluation

We explain the evaluation of Near operator using the 4-tuple “hasLocation Near Prof No”. First, we get the location of Prof from CAMUS. We developed our location ontology based on the space ontology developed by NASA [15]. The ontology provides tags to model the adjacency, containment relationship. The ontology also provides tags to model vertical spaces which are essential to represent floors.

```
<owl:ObjectProperty rdf:ID="adjacentTo">
<owl:ObjectProperty rdf:ID="contains">
<owl:ObjectProperty rdf:ID="outside">
<owl:ObjectProperty rdf:ID="inside">
<owl:ObjectProperty rdf:ID="above">
```

We retrieve the list of regions or rooms that are adjacent to Prof's room or region and then keep those services from the result set which are located in the same room of Prof or adjacent to it. If the resulting service set is empty after the short list, we reevaluate the services based on only the 4-tuples with Near operator having relaxation No. However, the meaning of near varies from application to application. Our current prototype cannot handle the various meaning of the near operator.

VII. Related Works

Lot of efforts have been put to make web services semantically enabled [11], [12], [13]. We bring those techniques to middleware based service environment. This paper is an extension of our previous paper [19] where we described the need for semantic context service discovery. In this paper we discuss on both context and device service. We include the Quality of Context attribute for context services and dynamic attributes for device services. We also incorporate some flexibility in service discovery so that the application needs less interaction with the middleware. In this respect, we investigated in middleware domain like CORBA, jini and JXTA. We have found one recent paper that tries to add dynamic attributes to CORBA services [20]. The authors were inspired by a similar implementation [21] on jini that could acquire dynamic attribute values from services periodically. But both of the works lack semantic service discovery mechanism. Our effort is to add semantic notations to jini services that can handle dynamic attributes and can also provide alternative or near by services, searching in the ontology semantically. In our implementation we use OWL-S with Jena which provides well accepted ontology and best suited reasoning mechanism for ontology, provides mechanism for service categorization.

VIII. Conclusion and Future Works

Ubiquitous computing envisions the accessibility of services from any where, any place. In this respect, a situation may occur that a user may switch to a different domain other than CAMUS and still want to access smart services of CAMUS. To support this kind of accessibility, we need to publish the services as web service and this motivates our next effort to define the mechanism to publish middleware services into the semantic web. The use of OWL-S to represent the services is our first step towards that goal.

References

[1] Intel personal server: http://www.intel.com/research/exploratory/personal_server.htm
[2] Q. Ngo Hung, Anjum Shehzad, Saad Liaquat Kiani, Maria Riaz, Kim Anh Ngoc, Sungyoung Lee: Developing Context-Aware Ubiquitous Computing Systems with a Unified Middleware Framework. The 2004 International Conference on Embedded & Ubiquitous Computing (EUC2004), Springer-Verlag Lecture Notes in Computer Science, August 26-28 2004, pp.672-681
[3] CAMUS Technical Report: http://oslab.khu.ac.kr/
[4] JINI: http://www.jini.org
[5] M. Román, et al.: Gaia: A Middleware Infrastructure to Enable Active Spaces. IEEE Pervasive Computing, Oct-Dec 2002, pp. 74-83
[6] Von Wolfgang Trumler, Jan Petzold, Faruk Bagci, Theo Ungerer: AMUN - Autonomic Middleware for Ubiquitious eNvironments Applied to the Smart Doorplate Project. International Conference on Autonomic Computing (ICAC-04), New York, NY, May 17-18, 2004
[7] Tao Gu, Hung Keng Pung, Da Qing Zhang: A service-oriented middleware for building context-aware services. Journal of Network and Computer Applications 28 (2005) 1–18
[8] CORBA: HTTP://WWW.CORBA.ORG
[9] Jxta: http://www.jxta.org
[10] OSGI: http://www.osgi.org
[11] Xiaocheng Luan: Adaptive Middle Agent for Service Matching in the Semantic Web: A Quantitative Approach. PhD Thesis, Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, UMBC eBiquity Publication, Nov 2004
[12] Lei Li, Ian Horrocks: A Software Framework for Matchmaking Based on Semantic Web Technology. International WWW Conference, Budapest, Hungary, 2003
[13] M. Klein, et. al.: Towards High-Precision Service Retrieval. IEEE Internet Computing, February 2004
[14] Feng Zhu, Matt Mutka, and Lionel Ni: Classification of Service Discovery in Pervasive Computing Environments, MSU-CSE-02-24, Michigan State University, East Lansing, 2002
[15] NASA Space Ontology: http://sweet.jpl.nasa.gov/ontology/space.owl
[16] OWL-S: http://www.daml.org/services/owl-s/
[17] Protégé: http://protege.stanford.edu
[18] Jena, A Semantic Web Framework for Java: http://jena.sourceforge.net/
[19] Maria Riaz, Saad Liaquat Kiani, Sungyoung Lee, Young-Koo Lee: Incorporating Semantics Based Search and Policy-Based Access Control Mechanism in Context Service Delivering. IEEE Fourth Annual ACIS International Conference on Computer and Information Science (ICIS 2005), 14-16 July 05, Jeju - Korea, pp.175-180
[20] Oussama Kassem Zein, Yvon Kermarrec: An approach for Service Description and a flexible way to discover services in distributed systems. International Symposium on Information Technology: Coding and Computing (ITCC), Vol (1) 2005: 342-347
[21] Choonhwa Lee, Sumi Helal: Context Attributes: An Approach to Enable Context-awareness for Service Discovery. Proceedings of the 2003 Symposium on Applications and the Internet (SAINT'03)