

# Polyarchical Middleware for On-Demand and Multi-Standard Services' Composition for Ubiquitous Computing

**Paper ID:** 57

**Contact Person:** Mengjie Yu

**Contact Address:** School of Computing and Mathematical Science  
Liverpool John Moores University  
Byrom Street Liverpool L3 3AF UK

**Email:** [cmsmyu@livjm.ac.uk](mailto:cmsmyu@livjm.ac.uk)

**Tel:** (44) 0151 231 2280

**Authors:**

*Mengjie Yu:* Research Student

*A. Taleb-Bendiab :* Professor

*D. Reilly:* Lecture

*E. Grishikashvili:* Research Student

Wail Omar: Research Student

**Related Topics:** Service Composition  
Open/dynamic grid service architectures  
Service & P2P/Grid Computing

**Key Words:** Runtime Dynamic Service Composition  
Polyarchical Middleware  
Service Interoperation & Adaptation  
Well-defined Service Description

**Abstract:**

Whilst the vision of on-demand computing is very seductive it engenders its own technical challenges including: design, development and deployment of ubiquitous utility services, and low-cost, low-skills and low-latency services re-assembly for lifetime management runtime due to complex and unpredicted service discovery, interoperation and adaptation. In this paper we argue for the need of a new model for on-demand ubiquitous services' activation through a polyarchical middleware, which enables on-demand composition of software applications regardless of service standards and middleware used. This paper will present early results of a research study into the development of a framework for ubiquitous service invocation/activation, which provides an abstract model for on-demand ubiquitous service composition and execution. This proposed polyarchical middleware could be used for on-demand wireless application service composition, which include ad-hoc service discovery, assembly using virtual containers, invocation and adaptation. The paper will finish with a critical review of our model and concluding remarks followed by an indication of further work

# Polyarchical Middleware for On-Demand and Multi-Standard Services' Composition for Ubiquitous Computing

M. Yu, A. Taleb-Bendiab, D. Reilly, E.Grishikashvili, Wail Omar

School of Computing and Mathematical Science  
Liverpool John Moores University  
Byrom Street Liverpool, L3 3AF UK  
{cmsmyu; a.talebbendiab; d.reilly; cmsegris; cmpwomar}@livjm.ac.uk

**Abstract.** Whilst the vision of on-demand computing is very seductive it engenders its own technical challenges including; design, development and deployment of ubiquitous utility services, and low-cost, low-skills and low-latency services re-assembly for lifetime management runtime due to complex and unpredicted service discovery, interoperation and adaptation. In this paper we argue for the need of a new model for on-demand ubiquitous services' activation through a polyarchical middleware, which enables on-demand composition of software applications regardless of service standards and middleware used. This paper will present early results of a research study into the development of a framework for ubiquitous service invocation/activation, which provides an abstract model for on-demand ubiquitous service composition and execution. This proposed polyarchical middleware could be used for on-demand wireless application service composition, which include ad-hoc service discovery, assembly using virtual containers, invocation and adaptation. The paper will finish with a critical review of our model and concluding remarks followed by an indication of further work

## 1. Introduction

Static distributed service composition has gradually presented limitations under a dynamically changeable network environment. The critical runtime usage requires distributed systems to be capable of automatically adapting and/or evolving their behaviours in response to any unpredicted changes such as: network faults, requirements changes and/or failed services. We feel that runtime services' composition can be more flexible than the traditional static approach. To best-fit end-users' needs, services can be composed/constructed at runtime from a set of available network and software services.

However, such dynamic composition requires addressing many technical issues including: seamless integration/interoperation, failure-detection and robust self-healing abilities. In order to response to such runtime requests, distributed middleware solution has been designed to bridge the gap between the distributed components and

on-demand service application. Through a higher-level programming abstract model, middleware architectures such as DCOM, CORBA, Jini, Web services, UPnP [1] provide different programming models for service composition and adaptation at both design and runtime.

However, such middleware technologies require that components must implement a single programming standard (e.g. Java Remote Method Invocation technology [2]) or adhere to a certain definition (i.e. CORBA Interface Definition Language [3]). If service composition occurs within the same standard, these distributed middleware can still flexibly respond to runtime on-demand requests. But considering the critical reality, service components may have been developed and deployed on different standards—even different operating systems. The current distribution middleware models render their unsuitable for service composition and adaptation between different varieties of system components. In short, such multi-standard service composition and activation would require a tremendous amount of manual configuration and adaptation. Ideally, if services of one system could be automatically discovered by another system and utilized, it would dramatically enhance system on-demand service composition and adaptation abilities at runtime. Without redesigning the system to accommodate the unpredicted changes, or even with less intervene from users, the distributed system can reconfigure/update the services without disrupting the system process.

Next generation of distributed architectures--Ubiquitous/Grid computing [4, 5] have been promoted to offer end-users a large scale virtual robust computing infrastructure. Within such a higher service composition model, components from different organizations and locations can seamlessly work together to solve a specific problem/request as mentioned in Open Grid Services Architecture (OGSA) [5].

In this paper, we describe a polyarchitectural middleware, which extends core middleware service models (of both Object-Oriented and Message-Oriented middleware) to provide on-demand multi-standard services' composition and adaptation across the ubiquitous computing.

Our model is based on the service-oriented architecture, in which service location transparency is addressed by services can be dynamically discovered and invoked by any other user application, components and/or software services whilst, the technical details of service composition and adaptation will be hidden away from end-users.

We address the challenge of runtime invocation adaptation to support service composition and self-healing strategies. Accompanied with well-defined service description and assembly language, dynamic configuration and self-adaptation can be exactly executed after precisely knowing about the environmental circumstances of service components.

We concentrate on enabling automatic service discovery and invocation, in which case data and service information are dynamically fitted into the invocation code. Human intervention is only essential to set up on-demand service requests and possibly involved to handle major changes in the environment or unexpected exceptions.

The remainder of the paper is organized into six sections: Section 2 describes several ongoing related research and development themes; Section 3, describes dynamic service composition solution, used to assemble on-demand service

applications; Section 4, outlines our proposed service adaptation strategy, used to compose service components and enable self-healing across varieties of standards/architectures; Section 5, considers the development of the Polyarchical Middleware architecture, which mediates the on-demand service requests and hides the technical details; Section 6 describes a recent case study, used for on-demand wireless application service composition; Finally, Section 7 draws overall conclusion and mentions directions for future work.

## 2. Related Work

The component-based software engineering community has widely used the connector and proxy model to model and abstract inter component interaction and invocation [6].

Web Service Invocation Framework (WSIF) [7], is designed to supply users with simple APIs to invoke Web services through abstract representations of the services. So far, WSIF provides a degree of protocol-independence, but however it assumes Simple Object Access Protocol (SOAP) as the underlying protocol. It allows developers/users to adopt own programming models without worrying about the service implementation regardless of SOAP packages used behind. However, our protocol-independence is designed to cross multiple service standards rather than SOAP.

Open Grid Services Architecture (OGSA), still in the early stage, has been introduced by IBM to integrate *Grid-enabled Web services* across the Internet. The transport protocol SOAP is used here to connect data and application on Grid service discovery, sharing and composition in a dynamic distributed environment. The standard interface of Grid service, described by Web Service Description Language (WSDL) [8], includes multiple bindings and implementations. Such Grid services can be deployed on different hosting environments-even different operating systems. As one of the invocation proxies, the existing WSIF has been embedded here to dynamically detect the SOAP binding protocols and construct the appropriate invocation code. Whilst, rather than only trying to deal with the Grid-enabled Web services, we have deliberately concentrated on ubiquitous services deployed on different service standards and core service middleware models (Object-Oriented and Message-Oriented middleware).

OpenWings [9] was first set up to meet the military use to discover replacement services in a critical and volatile environment. One of its components---connector, is used to represent the inter-communication between two different standards. By creating a protocol abstract layer, both sides need to implement a special APIs to translate method calls to and back from that layer, where all the data and information will be transmitted. Our work has similar goal but has focused on runtime service adaptation code and extra files generation. Both sides can still remain their original design without implementing those special APIs and sender and receiver proxies.

### 3. Dynamic Service Composition

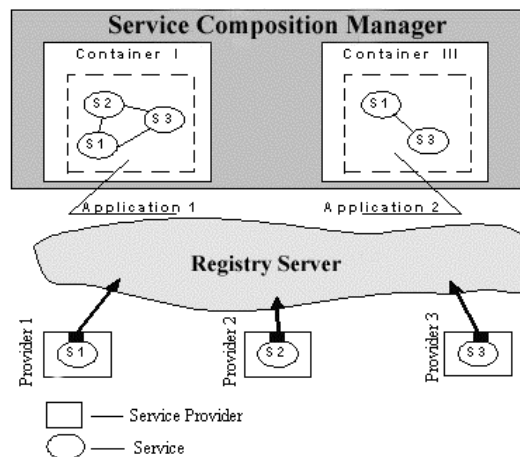
Unlike the traditional predictable and repetitive processes, end-users can runtime compose/assemble their own on-demand services from a range of services providers over the network. However, these services are likely to be developed and deployed in varieties of standards and middleware architectures.

In this paper, we introduce a dynamic service composition approach, which uses service virtual container and well-defined service description and assembly language to achieve runtime dynamic service composition and/or self-healing.

#### 3.1 Virtual Service Container

A *Virtual Service Container* is a virtual environment, which contains a composition of service-providing components and associated proxies. The *Virtual Container* acts as an assembly “workspace” or “workbench” in which components are assembled ready for composition into an on-demand application/service. Normally, each *Container* may contain one or more components, which may provide one or more services as shown in Figure 1. A *Service* is a logical concept, which could be hardware, software or a combination that may be distributed over the network. These *Services* may reside in a variety of existing *physical* containers (e.g. Java Virtual machine, Web server container, or simply JAR or ZIP files). The Service Composition Manager middleware service is responsible for the actual dynamic service composition. Any runtime service discovery, invocation and failure exceptions will be detected and dealt with by other managers in the proposed framework.

Fig. 1. Service composition manager



A *Service Provider* is a set of all necessary class files that enable service publication and service invocation over the network via a certain mechanism such as Java RMI, SOAP/RPC and/or SOAP/messages. Service providers enable services to

be discovered and accessed dynamically by software applications or other service providers.

In Figure 1, *Provider 1* offers *Service 1* by first registering it on a Registry Server—UDDI (Universal Description, Discovery and Integration), which was used to hold description of all published/available services encoded in Service Description and Assembly Language (SDAL). The latter<sup>1</sup> is an XML-based document containing information about the service, which are needed by software application or other service providers to access or invoke it. On the first invocation of a service method, the service comes to life within the *Virtual Service Container*. Later, more matched services can be managed to add into the virtual container on the demand of end-users.

### 3.2 Service Description and Assembly Language

Service Description and Assembly Language (SDAL) consists of two parts: service assembly language and service description language.

*Service assembly language*: is an XML file (Fig. 7). It is used for end-users to describe service components involved in the composition process at runtime.

Fig. 2. Service Description Language slip

```
<Service ServiceID="S1">
  <ServiceName>Clock</ServiceName>
  <ServiceDescription>http://localhost:8080/userdoc/Clock.wsdl</ServiceDescription>
  <ServiceCategory>Timer</ServiceCategory>
  <ServiceType>Research</ServiceType>
  <Container>cmsgris</Container>
  <ServiceLocation>cmsgris:3080/userdoc/Calculator</ServiceLocation>
  <ServiceBindingType>Java RMI</ServiceBindingType>
  <NumberOfUserInterface>1</NumberOfUserInterface>
  <InterfaceName>TimerSet</InterfaceName>
  <InterfaceLocation>http://cmsgris:3080/userdoc</InterfaceLocation>
  <NumberOfMethod>1</NumberOfMethod>
  <Method>SetTimer</Method>
  <Dependency>Antecedent</Dependency>
  <Authorized>True</Authorized>
  <ServiceContract id="SC1">
    <ServicesContractName>SCCmsgris</ServicesContractName>
    <ServicesContractLease>1/1/2004</ServicesContractLease>
  </ServiceContract>
  <AvailableLanguage>Eng,Fr,Ar</AvailableLanguage>
  <ServicesType>Research</ServicesType>
  <RequiredInfrastructure>
    <InfrID>I1</InfrID>
    <InfrID>I2</InfrID>
  </RequiredInfrastructure>
  <BelInfrastructure>True</BelInfrastructure>
</Service>
```

*Service description language*: we consider runtime dynamic service composition is a quite challenging issue, especially within such crucial time limits. As we found, it is hard to precisely predict what the exact environmental circumstances of service execution, and whether the process will be successful at composition time rather than design time. We also noticed that in the reality not every discovered service could be used as service composition components. Sometimes required on-demand service composition is significantly different from available service components. We feel it is important to have a well-defined service description for each component on both

<sup>1</sup> The difference between SDAL and WSDL will be discussed in the next section.

functionalities and even execution environment. It can greatly enhance runtime service discovery ability and find the best-fit components for service composition. Meanwhile, it can also reduce the system response time with less intervention from end-users. Figure 2, shows us an example of an XML-based *Service Description Language file*, which is used here to support runtime service discovery and composition.

Unlike traditional WSDL a SADL includes component's information not only service method name or message parameters, but also the infrastructure, which offers the operation container for software components. It uses a tag *ServiceDescription* (Fig. 2) to refer to an outside service description document, which can be encoded in WSDL or other available description technologies. The *RequiredInfrastructure* tag describes the infrastructure (Fig. 3), which provides the execution environment for the service component.

Fig. 3. Infrastructure description slip

```
<Infrastructure infrastructureID="I1">
  <infr_Name>Addition</infr_Name>
  <infr_Description>To do the addition process</infr_Description>
  <infr_Type>Software</infr_Type>
  <infr_Container>cmswomar</infr_Container>
  <infr_Host>www.jmu.ac.uk</infr_Host>
  <infr_Server>Apache/1.3.0 (Unix)</infr_Server>
  <infr_Platform>any</infr_Platform>
  <infr_Middleware>.Net</infr_Middleware>
  <infr_RequiredProcessor>PI 233MHz</infr_RequiredProcessor>
  <infr_RequiredMemory>8Mbyte</infr_RequiredMemory>
  <infr_RequiredStorageSize>1.4MB</infr_RequiredStorageSize>
  <infr_Contract id="IC1">
    <infr_Contract_Name>cmpwomar</infr_Contract_Name>
    <infr_Lease>3/7/2004</infr_Lease>
  </infr_Contract>
  <infr_Catogery>Addition</infr_Catogery>
</Infrastructure>
```

The details of service components on both software and infrastructure in SDAL can greatly help us discover/locate the right service components for composition at runtime.

#### 4. Service Adaptation

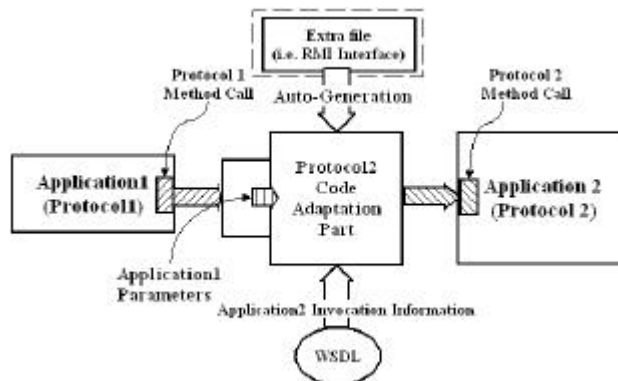
A successful service composition requires seamless service interoperation and well self-healing to recover from unexpected failure/changes. Both those strategies require robust service adaptation between different service standards to support at behind. In this section, we describe a service adaptation mechanism based on runtime code generation, which sets our approach different from other related works.

#### 4.1 Service Adaptation Mechanism

We begin by considering a scenario, in which two applications, based on different programming models (e.g. client-server/peer-to-peer) that need to communicate over a network. For such communication, our approach makes use of runtime auto-generated service adapters. The adapters can dynamically provide the necessary code adaptation to handle service interoperation and integration among different standards. The adapters make use of a template file, in which the invocation code for each individual standard has already been specified. Without implementing any special APIs, proxy or classes on both sides, these adapters set our work aside from that conducted by Openwings and WSIF. Service adapter allows the developers and end-users to ignore the details of the service adaptation and simply stick to their original implementation mechanisms. The following diagram in Figure 4 shows the anatomy of service interoperation sequence.

The protocol 1 method call can be reset to the normal protocol 2 mechanism after using the code adaptation. Application 2 can still be invoked by the protocol 2 method call in normal data encoding (i.e. serialization/deserialization in Java RMI or marshalling/unmarshalling in SOAP-RPC of parameters and results), which the other proposes including OpenWings intend to translate method calls into a standardized protocol transmission.

Fig. 4. Service interoperation sequences



Invocation information is retrieved by parsing a service WSDL and any extra file such as stub and/or proxy classes (i.e. Java RMI Interface) is also auto-generated at runtime.

#### 4.2 Adaptation Scenario

Here we consider two scenarios to demonstrate service adaptation behaviours. The first scenario represents default behaviour in that it attempts alternative invocations, regardless of service standard/protocol, without the need for end-user/system intervention. The second scenario considers how the user/system may specify

*particular preferences* relating to service standards or middleware types to be used in invocations.

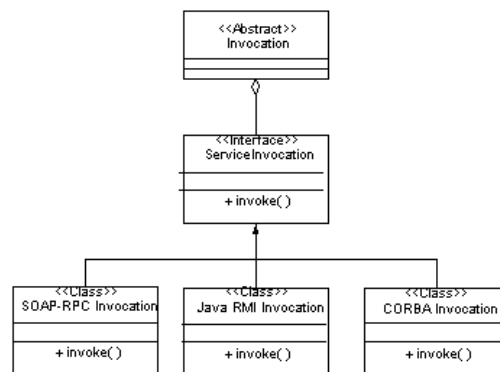
*Default behaviour:* Assumes end-users/systems have no particular preferences relating to the type of middleware technology or service standard used by the alternative service components. When a service is found (via lookup) in the registry server<sup>2</sup>, the Polyarchical Middleware initializes the invocation adapter template file to access/invoke any matched services as possible alternatives with the adaptation details hidden from end-users / systems. If a service invocation should fail, an exception will be caught and the next alternative component and associated connector will be sought and tried automatically, without end-user/system intervention.

*Particular Preferences:* allow the user/system to select specific service standard(s)/middleware technology(s), as options, which may serve as special runtime requirements. End-users/systems may still take advantage of the polyarchical middleware when it comes to the access/invoication of their own services. To do so, they first register their specific service and then allow the polyarchical middleware to take care of any adaptation of the invocation mechanism used to access/invoke their service.

### 4.3 Service Invocation Abstraction

Service Invocation Abstraction (SIA) is based on a set of interfaces and abstract classes, which are used to standardize and hide the exact invocation mechanism details (Fig 5). All the service adapter files implement the same interface, which declares a standardized *invoke()* method. Each adapter file can override the standardized *invoke()* method to implement their own specific service invocation details.

Fig. 5. Adapter class diagram



The end result of using a SIA is service protocol independence. At runtime, any adapter file can be initialized and cast to an instance of that interface, through which

<sup>2</sup> Universal Description, Discovery and Integration (UDDI) is used in this case.

the *invoke()* method will be called individually with a different invocation mechanism.

## 5. Framework Structure

The Sun™ Open Network Environment (Sun ONE [10]) first used the term *Polyarchival* to describe features of the next generation distributed architecture. Polyarchival middleware, smarter and with better communication, intends to support end-users to set up distributed applications in a higher-level model. Our own polyarchival middleware (Fig. 6) is based on a service-oriented architecture, which sits above the existing service middleware architectures. It can help end-users assemble/compose their on-demand services at runtime, without being tied to a particular service standard and/or provider. Instead, they can discover and adapt to a given service of choice from any available resource. This framework was designed to provide supports for service composition and adaptation through runtime code generation, thus abstracting service discovery and invocation details and thereby simplifying the service interoperation and integration process. Through this framework, end-users may concentrate specifically on their applications without having to worry about inconsistencies or mismatches between component standards.

**Fig. 6.** Polyarchival framework structure



- Service Call Dispatcher: provides access to this framework via Internet/Intranet. So far, a Java Tomcat HTTP server has been used to provide HTTP (TCP/IP) access to the polyarchival middleware framework. A Java Servlet container hosted by a Tomcat server is used to retrieve synchronous service invocation request through URL's from end-users. In our future work, a message-based asynchronous container will also be designed to receive requests from systems that may operate in a peer-to-peer fashion.

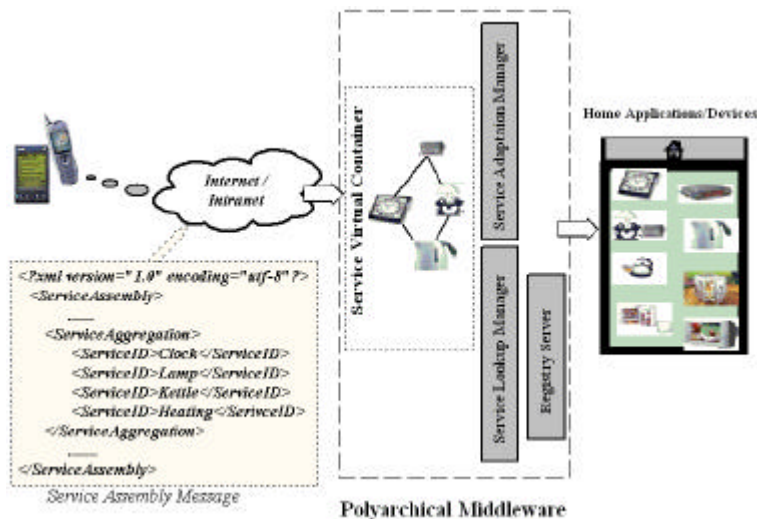
- Service Composition Manager: as mentioned before, is mainly responsible for runtime service composition after retrieving requests from service call dispatcher. It will initialize the virtual container for further runtime service composition. Requests of service components involved in the composition will be redirected to Service Lookup Manager for discovery/lookup appropriate components for execution. If request arises from adaptation between two components in different standards, it will pass this service adaptation request to Service Adaptation Manager. In addition, any failure will also be detected and handled by this composition manager.
- Service Lookup Manager: is used to discover/lookup appropriate components for service composition execution. The service lookup process comprises two separate processes: 1) It uses keywords, such as service name, to search inside the local UDDI Registry server and returns an array of matched service;
- Registry Server: To well support the runtime service discovery/lookup process, a UDDI compliant registry server was introduced, which holds description of all published/available services (the same service may provide different accessible bindings). Instead of encoding in traditional WSDL, the proposed the SADL is used to describe service components on both software and infrastructure level. Service lookup manager can discover most precise registered components through well-defined service descriptions.
- Service Adaptation Manager: is designed to handle any service invocation adaptation request occurred during the composition process. Runtime, the service adaptation manager can initialize the right adapter template file to switch invocation calls from one standard to another. The adapter template file contains an invocation mechanism, which has been indicated/mentioned in the SADL of each individual service component. Service invocation information for each individual component is parsed from corresponding WSDL files referenced inside the SADL. At runtime, they are appended to the adapter templates files for invoking the service. At this stage, the relevant extra files, which are used to assist the service invocation, are also automatically generated. For example, the service interface, will be required by a Java RMI-based service connector. The service adaptation manager will generate at runtime the associated RMI Interface by parsing the service description encoding using the WSDL specifications. Any failure due to changes in service components will also be detected by the polyarchical middleware at this stage and alternate components and connectors will then be sought.
- Extended Adapter: To make the polyarchical middleware more extendable, with respect to service interoperation and integration, we intend to introduce a Template Markup Language (TML) into the framework design. The TML will be used to add/describe more adapters as and when new service component standards are introduced in the future.
- Existing Middleware Standards and Service Providers: refer to the middleware technologies that provide normal middleware discovery/lookup and invocation protocols. Through the polyarchical middleware, service providers developed using these middleware standards may be integrated and interoperated by end-users.

## 6. Illustrative Example

In order to demonstrate the idea, an example of wireless service applications composition is used in this paper. We argue that current wireless devices can't fully utilize the network resources due to their own limitations such as: CPU performance, low memory. As a result, this proposed polyarchical middleware could be used to take over the runtime heavier lifting (e.g. ad-hoc service discovery, assembly using virtual containers, invocation and adaptation) from wireless devices (i.e. PDA or mobile sets). In order to illustrate the ideas above, we describe the following scenario:

*"A wireless device end-user is on his way back from work. Before he arrives at home, he wants to several of his home applications/devices to be set up ahead such as: set up the timer, turn on the lamp, switch on the Kettle and turn the heating to a certain temperature. Whilst, those networked home applications/devices are purchased from varieties of vendors, with different standards/drivers to make them work. Fortunately, the proposed polyarchical middleware comes to his rescue by receiving his on-demand request, composing these different devices and making them work together. All the details of service composition, discovery and adaptation will be hidden away from him."*

**Fig. 7.** Wireless assembly applications

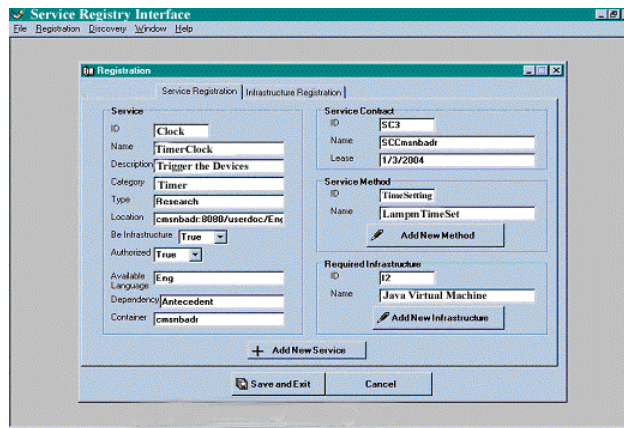


In this case, we assume all the home application devices are driven by the software to be networked. The software could be developed on different protocols such as: Java RMI, SOAP-RPC, and CORBA. Before utilizing these services, they are required to register on the local UDDI encoding in SADL through an interface (Fig. 8).

For precisely describing the on-demand service components, end-user sends an XML-based service assembly message (Fig. 7) to inform the polyarchical middleware. After retrieving this message, the service composition manager will initialize a service virtual container for further composition. The service lookup

manager is responsible for discovering all the required services (i.e. clock, lamp, kettle and heating) over home private network. Finally, the matched service components are added into the virtual container for composition in the sequence defined in that service assembly message.

Fig. 8. Service Registry Interface



The service composition manager is also in charge of monitoring the service composition process and detecting the service failure. For instance, the *Clock* component is used here to trigger the *Lamp* to switch on. Whilst, both *Clock* and *Lamp* have been developed in two different standards such as Java RMI and SOAP-RPC. However, at runtime, to still continue the process, the service adaptation manager can provide a SOAP-RPC invocation adapter (shown in following programming code slip) for smoothly switching the Java RMI call to a normal SOAP-RPC call. At runtime, a 'ParseWSDL.java' file is initialized by setting the parameter WSDLURL for the parser of the SOAP-RPC rpcrouter URL and serviceName(). As a result, this polyarchical middleware maintains the service composition process without having to halt for redesigning the component's invocation mechanism.

```
public class SoapAdapter {

    public void soapConnect(String wsdlURL) throws
    Exception

    {

        // Normal variable declartion

        ....

        // Runtime information filled in
```

```
ParseWSDL parseWSDL= new ParseWSDL(wsdlURL);

url=new URL(parseWSDL.soapURL());

Call call=new Call();

call.setTargetObjectURI("urn:Hello");

call.setMethodName(parseWSDL.serviceName());

call.setEncodingStyleURI
    (Constants.NS_URI_SOAP_ENC);

// invoke SOAP object

.....

}

}
```

## 7. Conclusions and Future Work

In this paper we have described an approach to dynamic service composition based on polyarchical middleware that enables end-users to discover, assemble and invoke on-demand service at runtime and accomplish critical service interoperation and adaptation tasks regardless of the technical details.

Development is still underway to extend the proposed polyarchical middleware runtime service composition ability on interoperation and adaptation. We also intend to extend its security mechanism and trust relationships between end-users and proposed polyarchical middleware.

## 8. References

1. E.Grishikashvili, N.B., D.Reilly,M.Allen,M. Yu and A. Taleb-Bendiab;School of Computing and Mathematical Sciences Liverpool John Moores University, *Automatic Computing: A Service-Oriented Framework to Support the Developement and Management of Distributed Applications*.
2. Edwards, W.K., *Core Jini (2nd Edition)*.
3. *Introduction to CORBA*.
4. *Ubiquitous Computing*, "<http://www.ubiq.com/hypertext/weiser/UbiHome.html>"

5. Liang-Jie Zhang, J.-Y.C., Researcher, IBM T.J. Watson Research Centre, *Developing Grid Computing Application, Part1, Introduction of a Grid architecture and toolkit for building Grid solutions.*
6. M. Yu, A.T.-B.D.R.W.O., *Ubiquitous Service Interoperation through Polyarchical Middleware.* 10. 2003.
7. Matthew J. Duftler, N.K.M., Aleksander Slominski and Sanjiva Weerawarana; IBM T.J. Watson Research Center, *Web Service Invocation Framework (WSIF).* August 9, 2001.
8. Erik Christensen, M.F.C., IBM Research; Greg Meredith, Microsoft; Sanjiva Weerawarana, IBM Research, *Web Services Description Language (WSDL) 1.1.*
9. Wade Wassenberg, S.E., Motorola ISD, *Protocol Independent Programming Using Openwings Connector Services.*
10. Shin, S., *Sun One: Now and Future.*