Anything is Service: Using LIR-OSGi and R2-OSGi to Construct Ubiquitous Service Network

Jinzhao Liu
College of Computer Science, Beijing University of Technology, Beijing, China
Email: innomentats@gmail.com

Dan Wang
College of Computer Science, Beijing University of Technology, Beijing, China
Email: wangdan@bjut.edu.cn

Yu Chen
Department of Computer Science & Technology, Tsinghua University, China
Research Institute of Information Technology, Key Laboratory of Pervasive Computing,Ministry of Education, Tsinghua University, China
Email: chyyuu@gmail.com

Yongqiang Lv
Department of Computer Science & Technology, Tsinghua University, China
Research Institute of Information Technology, Key Laboratory of Pervasive Computing,Ministry of Education, Tsinghua University, China
Email: luyq@tsinghua.edu.cn

***Abstract*—As an emerging portable service platform, OSGi is now taking a more and more important role in constructing a ubiquitous computing system. Since ubiquitous computing system is always a distributed system, people may face many challenges such as heterogeneous devices communication and vulnerable wireless network. This paper proposes a solution for constructing such a system based on LIR-OSGi and R2-OSGi. With the help of both frameworks, the system can treat any devices as service as well as cope with network exceptions automatically and transparently. This paper also introduces a demo system of smart house to prove our solution for the ease of use. Through this solution, any type of network devices can be connected to the network while the communication between devices remain robust.**

***Index Terms*—Ubiquitous Computing, OSGi, Middleware, LIR-OSGi, R2-OSGi**

I. INTRODUCTION

With the growth in the number and types of network devices, people are increasingly concerning about the interconnection between them. Mobile devices, such as laptops, PDAs, smart phones and Tablet PCs, which provide people with ubiquitous information services like Internet, Email, Social Network and LBS (Location Based Service), are most currently used among network devices. At the same time, these devices provide different functions as well. For example, notebooks provide powerful computing capability and input capacity with the help of keyboard; smart phones provide mobile network access, GPS and a large number of sensors [1], which screen are so small to enter words; Tablet PCs provide touch screen with considerable size which makes them suitable for applications with more touch. Through the interconnection among these devices, one can utilize the functions of others to make better use of different devices. For example, computers can control the smart phone to send messages or obtain location information; smart phones can take advantage of the computing power of notebook computers to complete complex calculations; people are able to edit emails on a Tablet PC (Tablet PCs screen size is usually far bigger than the smart phones, therefore it is more suitable for text editing), and then sent them through mobile network.

However, the interconnection and communication of devices becomes complicated due to the heterogeneity of the devices in ubiquitous computing network, such as the different hardware architectures, operation systems and programming languages [2]. Notebook is usually the X-86 architecture, but smart phone and Tablet PC are often based on the ARM architecture. Moreover, operation systems are often different such like Windows, Android and IOS, so developers have to write a lot of additional code to handle the communication between different platforms, which leading to increase application burden. Web-Service based Service-Oriented Architecture (SOA) is a good solution to solve the problem[3-4]. By encapsulating capabilities of device into service, applications with this feature can use any functions of other devices conveniently by accessing to the corresponding service.

SOA provides Interface definition specifications which are independent of hardware platform, operation system as well as programming language, so it can distributed

doi:10.4304/jsw.8.1.236-242

deploy, composite and apply loose coupled coarse granularity components through network based on demand. Different device functions can be encapsulated into services by language independent interface specifications. Meanwhile, complex functions can be completed by compositing diverse services.

Although SOA technology based on Web-Service has a good performance, it is still insufficient for ubiquitous computing environment [5]. In a ubiquitous computing environment, the devices are often resource-constrained; CPU performance and memory capacity are still inferior to PC. Therefore, we need a SOA framework which is more suitable for ubiquitous computing environments, which should take up less system resources and can provide better performance. Meanwhile, a well-designed SOA framework should hide network communication details to the upper layer and provide a transparent remote service invocation mechanism for the upper layer so that the upper layer application can transparently get access to remote services.

Moreover, in ubiquitous computing environment, devices are often interconnected through wireless network, which is unstable for weak wireless signal and interference by external source. At the same time, there are a large number of weak computing power devices with limited resources, which makes them incapable of handling excessively complex network exceptions [6]. For this reason, upper layer applications may crash and then threaten the normal operation of the system. In general, network-based applications deal with these exception at the application layer, this means developers need to consider not only software functionalities, but also network exception, which undoubtedly brings an additional work burden for the software development work. So we can complete this part of work in middleware, eliminating the need for the upper layer application to care. Exception handling code in network middleware is very easy to reuse, more stable and robust, in contrast, the code in application layer software is difficult to reuse.

We designed and implemented platform R2-OSGi [7] and LIR-OSGi[8] which are Service-Oriented middleware platform. In addition, a ubiquitous computing network oriented device interconnect platform, is completed, which is able to get transparently access and use service between diverse devices through encapsulating different functions into platform-independent and language-independent services. Our platform also provides automatic network anomalies handling capabilities which make the middleware system automatically handle the anomalies without application layers help when anomalies occur.

R2-OSGi is an improved platform for R-OSGi [9] by increasing the functions of network anomalies handling. It increases the detection of network anomalies, data backup on the network outages and data retransmission function on network recovery, thus can automatically complete the network exception handling without the need for user intervention.

R2-OSGi implemented a network anomaly detection module based on the heartbeat mechanism. By sending the heartbeat packet continuously, R2-OSGi can timely detect abnormal situation of the network layer. By loading the appropriate strategy, R2-OSGi can complete temporary processing on the anomaly data and actively try to restore the network connection. R2-OSGi takes good use of the dynamic and modular features of OSGi and describes the strategy as an OSGi Bundle, so that it can dynamically load and unload policy module.

LIR-OSGi is a distributed cross-language expansion based on OSGi, which remain s characteristics such as dynamic, modular and good performance, with distributed service invocation support and language-independent service invocation support added. This makes OSGi to be better used in the ubiquitous computing environment [10].

We designed OIDL (OSGi Interface Definition Language) which is defined for describing the service interface through a language-independent way by LIR-OSGi to implement service call in different programming environments [11]. On the platform, all the devices encapsulate their function into independent service, describe these interfaces by OIDL and then make them published [12]. These services are globally visible, any device can obtain and use the services to get interconnect and interoperate with each other, thus logically all equipment will be integrated into an overall.

Meanwhile, ODEX (OSGi Data Exchange) is defined by LIR-OSGi to complete data exchange between different devices. ODEX is platform-independent and programming language-independent data format based on JSON[13] therefore it is easy for user to read, lightweight and with highly efficient of parsing performance. JSON is more lightweight, carries the same amount of information by less space occupied and with better parsing performance compared with XML [14]. When it comes to the binary data, JSON is better to read, at the same time, has a better cross-platform performance.

The rest of this paper organizes as follows: The second section describes some relevant background including OSGI and R-OSGI, the third section presents the system architecture, the fourth section presents a real system based on our implementation, and the last is a summary.

II. OSGI AND R-OSGI OVERVIEW

OSGi is a dynamic module management system for java platform. It provides a complete and dynamic component model. Module and component, which is called Bundle in OSGi, can be installed, uninstalled, started and stopped dynamically without a reboot of the whole system. OSGi is a SOA-based platform which means bundles can interact with each other through services. It has a well-designed class-loading model. Each bundle has its own class loader to isolate with others. This ensures the independence and security when each bundle runs in an uninterrupted and independent environment.

Initially, OSGi is designed for embedded environment. Because the embedded environment is always a resource constrained environment, for instance, lacking of

computing power and little RAM storage, OSGi is implemented into a light weight framework. Therefore, OSGi has high performance and efficiency. But OSGi is a centralized framework and does not support distributed environment, which means all bundle should run in a single peer. Obviously this does not meet the requirements of pervasive computing.

R-OSGi is a distributed middleware platform that extends OSGi which is a centralized module management platform to interconnect modules and applications deployed in different devices. Through service agents, local modules can simply use remote services with the help of R-OSGi. The invocation to local service agent is converted into R-OSGi defined data form and then transferred to the peer where the service really locates.

After receiving the invocation data, R-OSGi will do the real service invocation and send the result to the caller. Whole procedure is completely transparent to the application.

Although R-OSGi extends OSGi to support distributed environment, R-OSGi itself is still an OSGi bundle, which means R-OSGi can only work on Java platform. In a ubiquitous computing environment, devices are always heterogeneous and not all devices can run JVM. So R-OSGi cannot solve the problem of transparent service invocation between heterogeneous devices.

Besides, ubiquitous computing environment is always connected through wireless netAnything is service: Using LIR-OSGi and R2-OSGi to construct ubiquitous service network work, for example,Wi-Fi, Bluetooth and Ad-Hoc. But an inevitable problem is the wireless network is unstable in many cases. The instability of network, such as network congestion, delay and disconnection, is an unavoidable problem. The R-OSGi based applications have to handle all the network exceptions by themselves because R-OSGi does not cope with any network problems automatically. This will bring much difficulty for application developers.

III. LIR-OSGI AND R2-OSGI DESIGN

A. LIR-OSGi

Our implementation LIR-OSGi is an extension of OSGi framework, which adds the distribution supporting as well as language-independence supporting. Thus with the help of LIR-OSGi, we can contribute a ubiquitous computing environment which mostly is a distributed system and all ubiquitous devices can be joint together by LIR-OSGi.

LIR-OSGi provides stable model for distributed service-oriented systems for developers to cope with heterogeneity and communications between two devices. It can transparently provide a RPC (Remote Procedure Call) and without increasing the programmer's workload by generating service broker locally and automatically.

LIR-OSGi defines a platform and programming language independent IDL (Interface Definition Language). Applications can describe a service interface via this IDL and compile it into a service implementation, then different applications written in different programming languages can use the same service.

In a ubiquitous computing environment, when two peers are connected, each peer will send the local service interface to the remote side. After dynamic compilation, the service proxy is generated and local applications can invoke remote services through local service proxy. This procedure is transparent to upper application. It is shown as Figure 1.

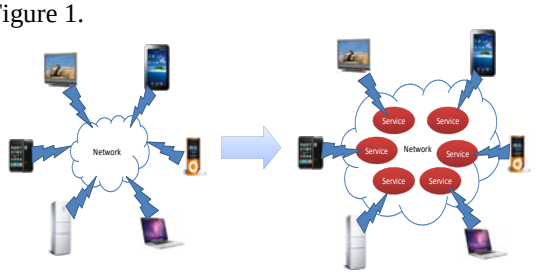

Figure 1. Service-Oriented interconnection of devices

The architecture of LIR-OSGi is shown as Figure 2.

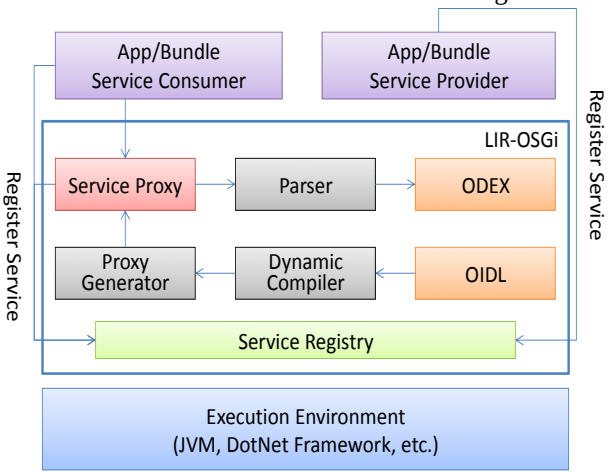

Figure 2. Architecture of LIR-OSGi

B. R2-OSGi

Our implementation R2-OSGi is an extension of OSGi to support automatic network exception handling. R2-OSGi can automatically handle network failures for all kinds of applications based on R-OSGi without changing their source code. The handling of network failure is completely transparent to OSGi application and easy to use and deploy.

R2-OSGi mainly consists of three parts: network exception detection module, network exception interception module and strategies. Network exception detection module can actively watch the status of network and detect the possible failures. When network exception occurs, it will inform network exception interception module to intercept the following operations of R-OSGi and inject the strategies to do some recovery work. The architecture of R2-OSGi is shown as Figure 3.

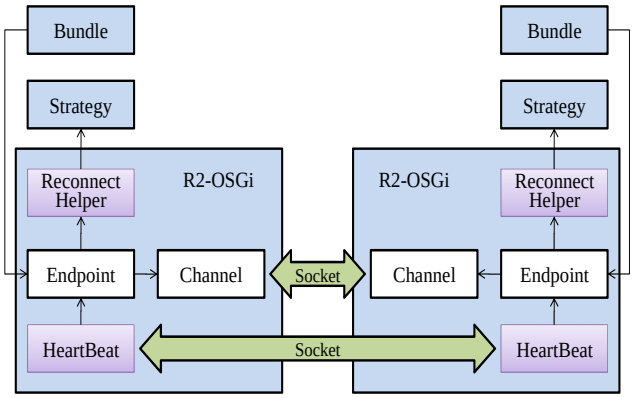

Figure 3. Architecture of R2-OSGi

IV. A SERVICE-ORIENTED MIDDLEWARE PLATFORM BASED ON LIROSGI AND R2-OSGI

A. Function Description

This paper designs a service-oriented middleware platform based on LIR-OSGi and R2-OSGi. This platform takes use of service model of OSGi and describes service interfaces through IDL defined by LIR-OSGi so that services can be invoked by programs written in different programming languages and deployed in different devices. Meanwhile, this platform takes advantages of R2-OSGi to automatically handle network exceptions, making applications can focus on their own logic without the need for coping with network layer issues. In this platform, all the functions of the devices are registered as OSGi services and can invoke each other with the help of LIR-OSGi. The architecture of this platform is shown as Figure 4.

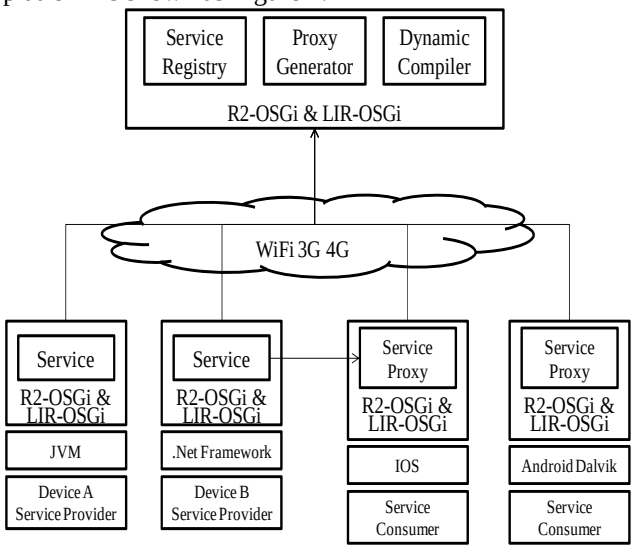

Figure 4. Architecture of the platform

There is a service registry existing in our platform. It retains all the service status information. When a device needs to use a service, it first has to establish a connection with the Service Registry to obtain status information of all the services, and locate the service it needs. Then it will start a connection to the service. After the connection is established, LIR-OSGi will send the service interface description information back to the requester side, and then call the dynamic compiler to generate the service proxy. Service proxy works at the requester side, responsible for converting the service invocation to the LIR-OSGi defined ODEX form, and then LIR-OSGi will send the ODEX to the service provider, make the real service invocation and send the result back to the caller.

When network failure occurs, R2-OSGi will intercept the data sent to the remote side and send it to the strategy module, and then R2-OSGi tries to reestablish the connection.

Once succeed, R2-OSGi will inform the strategy module and the later one will try and process the data which are intercepted by R2-OSGi. Some important process is described as follows.

B.Service Binding

Because the caller does not know the services existing in the network and their locations, so there must be a node keeping all the information of services. This node is called Service Registry, deployed in a specific device as a service. Any service provider should summit the service registration information to the Service Registry and notify the Service Registry when service status changes such as service updates, service stops, etc.

When an application requests a service, it first needs to connect to the Service Registry and obtain the current service status information. After find the required service, it then will establish a connection to the service provider. And the service provider will send the service interface description information to the requester once the connection is established.

LIR-OSGi generates the service proxy using the service interface description information. Then the application can use this service proxy to invoke the remote service. The binding process is shown as Figure 5.

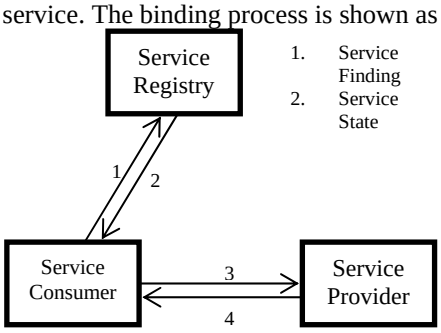

Figure 5. Process of Binding

C. OIDL

In OSGi, Services are specified by Java interfaces. Bundle can register services to Service Registry by implementing these interfaces. But interface definition written in Java cannot work in other programming languages. So it is impossible to generate a Service Proxy in a specific language with an interface definition written in different ones. We need a language-independent interface definition for all the languages to

define the service.

So we introduce OSGi Interface Definition Language (OIDL) to describe an interface with which a service can be implemented and registered. OIDL is syntactically like JAVA and C++ and only has a little difference. An example for interface definition written in OIDL is like:

interface oidl.example.IServicef
integer getStatus();

This definition specifies an interface named oidl.example.IService. This interface supports one operation called getStatus which takes no parameters and returns an integer (in Java and C++, it is int). After translated into Java or other programming language code, it can be used as a normal interface (in C++ it is class).

The most important feature of OIDL is language-independence, which means OIDL should be a declarative language, not a programming language. It must be translated into the specific language when it is used to implement and register a service. What's more, the data type in OIDL should be accepted by all potential programming languages, so it cannot carry the data type which is not supported by all languages or complicated.

After this interface is translated into Java, it seems like:

package oidl.example;
interface IServicef
int getStatus();

D. Generation of Service Proxy

Service Proxy is generated by Proxy Generator automatically. Once LIR-OSGi receives the OIDL information, Proxy Generator will call the Dynamic Compiler to parse the OIDL, and generate the service proxy according to the location. Service proxy is basically an implementation of the service interface. It takes charge for packaging the arguments of the service invocation, starting a RPC and waiting for the result. After the result is returned, service proxy will send it to the service caller.

After generating service proxy, LIR-OSGi will instantiate it. Then it will be registered with the local OSGi framework, so the local application can use this service proxy as a local service.

When local application calls the remote service through service proxy, service proxy will convert the invocation information into a language-independent representation ODEX.

ODEX will be sent to the remote peer and used to do the real service invocation. Then the result is transferred to the caller in the same form of ODEX.

E. ODEX and RPC

In order to exchange data between different programming languages, we introduce ODEX (OSGi Data EXchange). ODEX is a data format to transport data between two peers. It is based on JSON, which is a lightweight data-interchange format and easy for users to read and write as well as easy for machines to parse and generate. The reason we choose JSON rather than XML is that the performance of parsing and generating JSON is better. And the reason we do not choose binary is that JSON has a better readability and platform independence.

Two types of data ODEX need to carry. The first one is service invoking data and return data. Service invoking data contains the services name, invocation UID, parameters,parameters types and return type, and the return data includes invocation UID, return type and return value. Here is an example for the ODEX data of invocation of the service which will be illustrated next.

{"ServiceName":" oidl.example.IService",
"MethodName": "getStatus",
"ParamType": [ ],
"ParamValue": [ ],
"ReturnType": "integer"
}

The second one is service registration information including new services adding, service modifying and service removing information. Normally, it contains the OIDL speci fied interface and the location of the service. Any services, who want to be recognized as a remote service provider, should put a specific parameter in the LDAP filter while registering to Service Registry. Once two peers are connected, LIR-OSGi will automatically build service proxies for these services by exchanging service registration information.

After bound to the remote peer and created service proxy, the local service now can invoke the remote service by starting a RPC. The PRC is started by the calling of local service consumers to the service proxy.

The services method which is implemented by out compiler, marshals the request. Then the marshaled request data is sent through prebuilt TCP connection. When the remote peer receives this request, it restores the request data and does the actual service invocation using Reflection (in Java and C#). The result of invoking the service is then marshaled in the same way. After that the result is send back to the caller and the RPC is finished.

F. Network Exception Detecting

Heartbeat mechanism is one of the most common ways to provide high available network.

It uses continuously sent package, which is known as heartbeat package, to constantly test the network status. The client continuously sends heartbeat package in order to inform the server that the network is still available.

Within period of time has not received the palpitation package, the server could deem that the network is broken down, or the client is offline.

In R2-OSGi, we use two-way heartbeat package, that both ends of any connection are sending and receiving the heartbeat packets at the same time. A certain time interval T0 is between two packages. When either end of the connection has not received the heartbeat package for N (N >1) cycles, the network can be considered to be broken down.

In different network environment (LAN, Wi-Fi, Bluetooth, etc.), T0 and N can be set differently according to different network attribute values, for example, delay, bandwidth, etc, so that R2-OSGi can be adapted to different network environment. Furthermore, T0 and N can be set to the most reasonable parameters based on empirical data, which can make R2-OSGi to

detect the presence of network failures in the shortest possible time.

G. Apply Various Strategies

Due to different network environments and applications, the actions taken for handling network exception are often different. For example, a client sends several requests for the current time to the server before the network breaks down. Unfortunately, only the last request should be resent after the network recovers. But if it is such a request as printing, the entire request sequence should be restored and resent after the connection rebuilt. So, R2-OSGi should be able to take different action for different applications and network environments. In R2-OSGi, the action taken to handle network exception is called strategy.

A strategy must implement the interface Strategy. This interface is defined as follows.

```
public interface Strategy
{
void onDisconnect();
void incomingMessage (RemoteOSGiMessage message);
void onReconnectSucceed();
void onReconnectFailed();
  }
```

All the methods are invoked by Reconnect Helper. When Network Exception Detector detects the network exception, method onDisconnect will be called. After that, when Endpoint tries to send Message via Channel, the Massage is intercepted by Reconnect Helper, and sent to Strategy Module through calling the method incomingMessage. When the network is successfully rebuilt, method onReconnectSucceed will be called, the strategy which user defined previously will be applied. On the contrary, Method onReconnectFailed will be called.

Except the function describes above, another function of Reconnect Helper is selecting an appropriate strategy. User should prescribe the rules for R2-OSGi to choose which strategy will be applied. When R2-OSGi detects network exception, it will check the rules user provided, and then choose the corresponding one to deal with network failures. After that, all the Messages Endpoint tries to send to the remote peer via Channel will be sent to the selected Strategy.

V. EXAMPLE

In this section, we demonstrate our system through a specific example. This system is a virtual smart house prototype system including lights, TV, air conditioning and other virtual devices as well as the location information of users. In this system, all devices are packaged as services, so that they can use each other's functions by service invocation. We also implemented a reasoning module which controls status of each device according to the location changes. All devices and their status are displayed through a simulator. By using a simulator, location of users can be simply changed and all devices will respond. The interface of the simulator is shown as Figure 6.

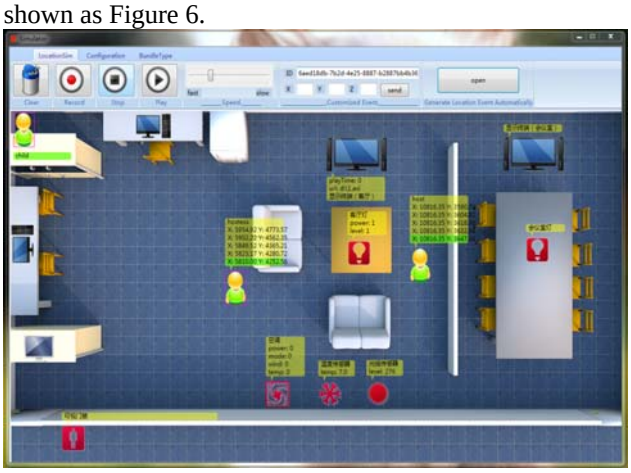

Figure 6. Interface of the simulator

This system is a real distributed system. The two lights locate on a laptop, the air conditioning is an ARM board and the two TVs are two laptops. Service Registry runs on a laptop along with this simulator. All the devices are connected via Wi-Fi. This system deploys as Figure 7.

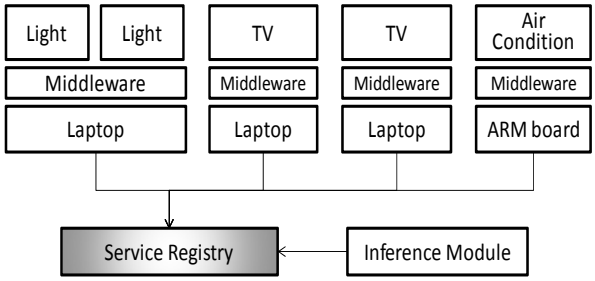

Figure 7. Deployment of the demo system

When ones location changes, the reasoning module will calculate the new status of the devices. Then it will invoke the devices service to change the status. As a feature of LIR-OSGi, reasoning module need not concern about where the service locates and the programming language used by the service provider. It just invokes the service with the language-independent service interface. It shows as Figure 8.

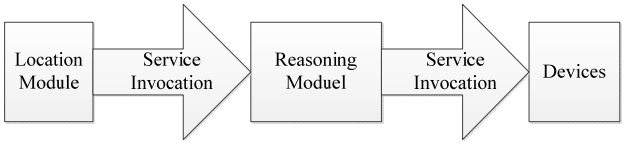

Figure 8. Service Invocation Process

For example, we can control the status of the lights through reasoning module. In the reasoning logic, when the user enters the living room, the light in the living room should be turn on; when he or she leaves, the light should be turn off. In this example, the reasoning module presents as a virtual module, providing a service interface for updating the location of the users. When ones location changes, the locator sends the new location to the reasoning module through the service interface. Then the reasoning module controls the status of the light by determining whether the user is in the living room.

Through this example, we can see that the interaction between devices is done by service invocation through language-independent service interface. The application need not concern the platform the service runs on and the programming language the service is written when invoking the service.

VI. Conclusion

This paper introduces a solution for building a ubiquitous computing system based on our implementation LIR-OSGi and R2-OSGi, which are extensions of OSGi framework to provide the distribution supporting as well as language-independence supporting. By taking advantage of LIR-OSGi, applications can make service invocation while not have to concern the hardware platform and programming language. Heterogeneous devices can be joined together through LIR-OSGi to construct a ubiquitous service network. With the help of R2-OSGi, the system can automatically cope with network exceptions, so that the exceptions of network layer can be transparent to upper layer applications. It make the system more flexible to handle network exceptions can the whole system can be more robust. Through this solution, any kind of devices can be easily connected together to construct a service network. Even for legacy system, devices can also be connected to the network by packaging them into services.

Acknowledgment

This research is supported in part by a grant from Beijing Municipal Natural Science Foundation of China (Grant No. 4122007).

References

[1] JasonWiese, Patrick Gage Kelley, Lorrie Faith Cranor, Laura Dabbish, Jason I. Hong, John Zimmerman. Are You Close with Me? Are You Nearby? Investigating Social Groups, Closeness, and Willingness to Share. Proceedings of UbiCom'11. 2011:197-206

[2] An Architecture of Mobile Web 2.0 Context-aware Applications in Ubiquitous Web. Journal of software. Special Issue: Middleware and Network Applications. Vol 6, No 4.2011:705-715.

[3] Andr Bottaro, Anne Grodolle. Home SOA - Facing Protocol Heterogeneity in Pervasive Application. ICPS'08, Italy, 2008:73-80

[4] Mohamed Adel Serhani, Abdelghani Benharref . Enforcing Quality of Service within Web Services Communities.Journal of Software, Vol 6, No 4 .2011:554-563.

[5] Hayyan R. Sheikh. Comparing CORBA and Web-Services in view of a Service Oriented Architecture. International Journal of Computer Applications. 2012, 39(6): 47-55

[6] Young-Woo Kwon, Eli Tilevich and T.Apiwattanapong, "DR-OSGi: Hardening Distributed Components with Network Volatility Resiliency", Middleware 2009

[7] Jinzhao Liu, Yongqiang Lv, Dan Wang, etc. Flexible, plug-and-play network middleware against network instability with R2-OSGi. 5th International Conference on Pervasive Computing and Applications (ICPCA), 2010: 244-249

[8] Jinzhao Liu, DanWang, Yu Chen, etc. LIR-OSGi: Extends OSGi to support distributed and heterogeneous ubiquitous computing system. 6th International Conference on Pervasive Computing and Applications (ICPCA), 2011: 169-174

[9] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe, R-OSGi: Distributed Applications Through Software Modularization, Middleware 2007, LNCS 4834,2007

[10] Seung Keun Lee and Jeong Hyun Lee, "OSGi based service mobility management for pervasive computing environments", IASTED, 2006

[11] Jouve, W., Lancia, J., Palix, N., Consel, C., Lawall, J. High-level Programming Support for Robust Pervasive Computing Applications. Pervasive Computing and Communications. 2008(4): 252-255.

[12] Q Limbourg, J Vanderdonckt, B Michotte. Usixml: A user interface description language supporting multiple levels of independence. Engineering Advanced. 2004(12): 241-256.

[13] Introducing JSON. http://www.json.org/

[14] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, Clemente Izurieta, Comparison of JSON and XML Data Interchange Formats: A Case Study, Computer Applications in Industry and Engineering (CAINE), 2009: 157-162

Jizhao Liu was born in 1986. He received the M.S. degree in Computer science and technology from Beijing University of Technology in 2012. His research interests include pervasive Computing and middleware.

Dan Wang was born in 1969. She received the Ph.D degree in computer software from Northeastern University in 2002. She is a professor at Beijing University of Technology. Her research interests include trustworthy software, distributed computing, etc.

Yu Chen was born in 1971. He received the Ph.D. degree in Computer Science and technology from National Defense University in 2000. He is an associate professor at Tsinghua University. His research interests include system software, pervasive computing, etc.

Yongqing Lv was born in 1979. He received the Ph.D. degree in Computer Science and technology from Tsinghua University in 2006. He is an assistant professor at Tsinghua University. His research interests include system software, pervasive computing, etc