Towards Reliable OSGi Framework and Applications

Heejune Ahn
Seoul National Univ. of Tech.
Nowon-gu, Seoul,
Republic of Korea
+82-2-970-6543
heejune@snut.ac.kr

Hyukjun Oh
Kwangwoon Univ.
Nowon-gu, Seoul,
Republic of Korea
+82-2-940-5132
hj_oh@kw.ac.kr

Chang Oan Sung
Indiana Univ.
Southeast, Albany, IN
USA
+82-2-521-1520
cosung@ius.edu

ABSTRACT
Upcoming ubiquitous computing systems are required to operate in dynamic, diverse, unveri fied, and unpredictable operating environment. The OSGi (Open Service Gateway initiative) framework employs the service-oriented approach and the java classloader architecture for the runtime service deployment, that are well suited to the dynamic environment envisioned for home networking and ubiquitous computing. However, the current OSGi framework does not provide full reliability measures, especially for failure conditions such as network, device, and application failures. This paper analyzes software reliability issues in OSGi framework and proposes a proxy-based reliable extensions. The design concept is implemented and partly tested on an open source OSGi platform, Oscar, for the smart home residential gateway test-bed.

Categories and Subject Descriptors
D.4.5 [Operationg Systems]: Reliability – backup procedures, check point/restart, fault-tolerance, verification.

General Terms
Design, Reliability, Standardization.

Keywords
OSGi, reliability, proxy-based, service oriented architecture (SOA).

1. INTRODUCTION
Upcoming ubiquitous computing systems are required to operate in dynamic, diverse, unveri fied, and unpredictable operating environment. In order to adapt such a highly dynamic operating environment, the service-oriented approach/architecture (SOA) concept is founded. SOA concept is based on service interface descriptions, service registrations, and service discovery functionalities. Recent computing standards and systems like OSGi, Web Services, MS.NET, and CORBA services, follow the SOA concept [1, 3, 14].

In the paper, we propose a proxy-based fault tolerant approach for SOA systems. The OSGi service platform is investigated but same reliability issues can be applied to a general pervasive service oriented computing systems. In spite of traditional fault tolerance measures in computing systems, the SOA approaches brings out new challenges in software development around service discovery, service registration, service deployment, service processing, and service security. One of issues that are not fully addressed in OSGi framework is dynamic service management for reliable and fault-tolerant systems. Recent researches have put focus on service management and application level context-awareness [4,5,6]. However, there is almost no discussion on reliability and fault tolerant issues on OSGi framework. The paper presents the proxy based reliability extension for OSGi framework. The proxy wraps a real service object, intercepts calls from client service object, routes the request calls to the best service at runtime, and handles failures that arise. Furthermore, proxy approach can provides extensive status monitoring, dynamic reconfiguration, and fault handling with no explicit application codes. Simply put, the proxy architecture accomplishes the loosely coupled principle for the current OSGi framework [2].

The remaining part of the paper is organized as follows. Section 2 describes the reliability issues of OSGi frame including existing reliability measures and risks. Section 3 presents our extension for reliable OSGi framework based reliability measures for the current OSGi framework and describes the implementation approach. Section 4 summarizes our implementation and application experience. We also discuss differences of our approach against the previous works, and suggest the future working topics. Section 5 concludes our work.

2. RELIABILITY ISSUES of OSGi
The OSGi framework defines the execution environment for application and utilities services. The framework provides minimal functionalities including component management, service registry, and java class loading. The services are represented as a java interfaces that is a format description of service functions. Service implementations are packaged into a ‘bundle’, which is a jar archive file with OSGi extended manifest file. The manifest headers in the manifest file provides to OSGi framework information of the bundles, which includes required packages Bundle-Activator, Bundle-ClassPath, Bundle-Required, Export-Package, Import-Package and so on.
All OSGi bundles go through life cycles of null, installed, resolved, starting, stopping, active, uninstalled. Bundles are downloaded and installed. Figure 1 illustrates the OSGi bundle life cycle. A bundle can be installed and uninstalled while the framework is running, i.e., a hot deployment. When a bundle is installed it is assigned a unique id by framework. For the installed bundle to run, the java package dependency should be met first, which is named resolved state. Then bundle can be activated when the framework calls 'BundleActivator.start()' method. BundleActivator should handle all system related works, so that the applications logic includes only the application layer specific codes. The system related works includes registering and unregistering its own service, and find required services. Because multiple service implementations can exist on one platform, LDAP string is used for handling the preference selection. The system dependent operations provided by the OSGi platform are done through a context object, which is assigned when the bundle is activated. Since all the interaction between software components occurs via service, service management, in particular service dependency management is the key function of OSGi framework. The current OSGi specification version 3.0 provides service registration, service discovery, service preference, service binding, service tractability.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'06, April, 23-27, 2006, Dijon, France.
Copyright 2006 ACM 1-59593-108-2/06/0004...$5.00.

3. RELIABLE OSGi DESIGN

We have noticed that the difficulties in providing reliability in OSGi framework are mainly due to its limited support of SOA’s loose coupling and late binding. Once an OSGi required service is bound, the service is normally accessed until either the user bundle or the service provider service is unregistered. To be fully loosely coupled, the provider service should be selected at use time, i.e., when the user service calls the service. In order to handling the fault identification, isolation, and notification, and recovery in OSGi frame, we added a proxy-based extension to OSGi framework. The proxy [7] provides change for the framework to intervene between two services. OSGI spec 3.0 also provides an indirection mechanism using ‘org.OSGi.framework.ServiceFactory’, but the main purpose of the ServiceFactory class is to provide customized service. Moreover, once assigned package will continue to service the bundle until it is uninstalled at last. In contrast, our design allows dynamical changes during service whenever it is possible. Note that some service build service context while the bundles interacts.

Figure 2 illustrates the components of OSGi framework with focus on reliability extension. Four components are added, proxies, policy manager, dispatcher, and monitor. Proxy is constructed for each service instance. The dispatcher arbitrates and routes service call to best service implementation with the help of policy manager. The monitor performs status checking of each service instance. These components relays on the existing OSGi functionalities, service registry, package admin (classloader), and bundle management.

The simplest application case is when the service provides stateless service. In the case, the full level dynamic selection is possible. However, we do have condition, stateful service is required. Our work extends service for those stateful services with context. Whenever the state information is required, service context is transferred to provider services. However, to fully provide fault tolerance to stateful service, persistence and recovery mechanism is typically required, which is not fully considered in our work. From our smart home application experience, most of service can be implemented as stateless service or simple context information, i.e, a set of several primitive values.
Figure 2. The Proxy-based reliable OSGi platform, an extended Oscar

3.1 Fault detection: Proxy-based Fault Monitoring

Efficient fault detection is a tough challenge in original OSGi framework. There is no systematic frame support that a client service can check the health of provider services before using it. The provider service has to check its own status and unregister itself when it cannot provide its service properly, and the client service has to listen to all registration events. It must be quite cumbersome and error-prone. In contrast, the proposed architecture provides both active and passive fault monitoring mechanisms. Figure 3 shows a simplified code for the proxy object. A proxy intercepts calls to a service implementation, and it can monitor the status of the service, such as success or failure of call and service execution time (Figure 4). The history of execution results can be used for performance of the service implementation. Whenever it is possible and required, the monitor makes active check on the health of each service. This active monitoring function is really useful when the service availability is time-varying. Since OSGi operating environment is highly dynamic as other pervasive computing environment, the availability of service is time varying. For example, temperature sensors located in living room, kitchen, bedroom In this paper, the OSGi platform will refer the operator description and bundle description to generate monitoring rules, and according to the monitoring rules measure the service and notifies the system failure to definition.

```java
public class MonitorProxy implements InvocationHandler {
    private Dispatcher dispatcher; //master
    private Object realsvc; // real service
    private List histErr; // history of errors
    private MonitorProxy(Object obj, Dispatcher disp) {
        this.Realsvc = obj;
        this.dispatcher = disp;
        histErr = new ArrayList();
    }
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        dispatcher.invoke(proxy, method, args);
    }
    private Object internalInvoke(Object proxy, Method method, Object[] args) throws Throwable {
        try{
            return method.invoke(obj, args);
        }catch{InvocationTargetException e){
            histErr.add(method);
            throw e.getTargetException();
        }
    }
    public static synchronized Object proxyFor(Object obj, Dispatcher dispatcher) {
        Class objClass = obj.Class();
        return Proxy.newProxyInstance(
            objClass.getClassLoader(),
            objClass.getInterfaces(),
            new MonitorProxy(obj, dispatcher));
    }
}
```

Figure 3. Service Monitoring Proxy pseudo code: the proxy intercepts the call from service users, then check the response results..

Figure 4. Typical service call-flow: The vertical thick line represents call duration. The box is when the first call to service A is not satisfied, for example exception occurred. The dispatcher retries the same call to alternative (less preferred) service.

Efficient fault detection is a tough challenge in OSGi platform. Despite OSGi follows the SOA concept, the binding operation is rather static. For example, once the bundles obtains the access to a service or resource. Another serious problem in the static control is that a service can overuse system resource.
3.2 Fault Isolation: Dependency Graph and Automatic Reconfiguration

Once a fault and the erroneous service is identified, the bundle should be isolated by stopping and if it is permanent failure, uninstalling the service. Otherwise, the failure propagates to other services soon, so that containments overall system. Fault isolation in our design, the fault monitor checks the containment with service dependency descriptor. When a service instance is assigned to a client service, fault monitor construct dependency graph, which is simply a Java List object. When the service is found to be failed, all the referencing bundles is to be checked immediately for the containment. If it is also contained, the bundles also are stopped and uninstalled if required. This process is repeated to the top bundles, which are usually application bundles. The containment detection and isolation algorithm is summarized as follows:

Step 1) find failed service and bundles (fault identification)

Step 2) check if the failure is temporary (put it the recovery monitoring list) or permanent (free references and uninstall service bundle).

Step 3) find service bundles that refer the failed service bundle (dependency check).

Step 4-a) If no more referring service, stop the fault propagation check.

Step 4-b) If it has dynamic dependency and can find alternate service instance for the cardinality, stop the fault propagation check.

Step 4-c) O.W. stop and uninstall the bundle, go to step 2

Figure 5 provides a simple service dependency graph and fault isolation example. The scenario has two top-level application services T1 and T2, which uses middle utility service M1 and M2, respectively. M1 service uses one Service A and one service B. M2 uses one service B and one service C. The typical bottom services are network device service, sensor device service, and actuator services. Also assume that service B is stateless service so one instance can serves multiple users simultaneously. A typical example for service A is network service that has multiple Internet connections. In Figure 6, service A has two failed service instances of three, but it can still provide service. But the service C has one instance and it is failed, so no service C is available at the moment. The reference graph shows the M2 and T2 cannot operate properly due to the required services’ failure, while the M1 and T1 can operate without interruption. Imagine what happens and how it can be disastrous when the T2 service operating without M2 service.

Figure 5. Fault propagation detection and isolation

To declare service dependency, we borrowed and modified an XML descriptor in [6]. We made a few modifications. The bind and un-bind method attributes are not used since it is not utilized in our approach. Values for policy attributes are extended to stateless, stateful, and static. Stateless is equivalent to dynamic and static is to static in [6]. The stateful policy is for the service that uses the service context object. Finally we added fault element, which is dealt in subsection 3.4. Table 1 is a brief summary of the XML elements and Figure 6 is an XML instance.

Table 1. XML elements for Service dependence descriptor

<table>
<thead>
<tr>
<th>XML</th>
<th>O/M</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bundle&gt;</td>
<td>M</td>
<td>Top level elements for bundle, deploy unit</td>
</tr>
<tr>
<td>&lt;instance&gt;</td>
<td>M</td>
<td>Class name for Service instance</td>
</tr>
<tr>
<td>&lt;provides&gt;</td>
<td>O</td>
<td>Interface name for service that this bundle provides</td>
</tr>
<tr>
<td>&lt;property&gt;</td>
<td>O</td>
<td>Properties which is used for preferred service selection</td>
</tr>
<tr>
<td>&lt;requires&gt;</td>
<td>O</td>
<td>Interface that this bundle uses with the following attributes. Service : service name Filter: LDAP filter string (preference) Cardinality: required number of service policy: stateless, stateful, static (binding property)</td>
</tr>
<tr>
<td>&lt;factory&gt;</td>
<td>O</td>
<td>Class name for Factory for service object</td>
</tr>
<tr>
<td>&lt;fault&gt;</td>
<td>M</td>
<td>Declare what to do when the service cannot not exceute [subsection 3.4]</td>
</tr>
</tbody>
</table>

```xml
<bundle>
  <instance class="org.foo.impl.FireMonitorImpl">
    <service interface="org.foo.service.FireMonitor" />
    <property name="version" value="1.0" type="string">
      <requires>
```

---

"Health service  Containment service  Failed service"
3.3 Fault Recovery

Once Fault is identified, the recovery procedure is based on the redundancy of same service. The fault monitor request a new service proxy available to the policy manager, and the policy manager will provide a service object. One non-trivial issue we conflicted is service context management for stateful services. We defined service context for client service object, and set it to ‘java.lang.ThreadLocal’. We might use the context object as an input argument of service call, which requires service interface convention is changed so that the client service bundle have to be reprogrammed. In both cases, the provider service should be reprogrammed in order to fully utilize our dynamic proxy based approach. When changes are impossible, the service should be declared as ‘static.’ Also as mentioned above, to fully provide fault tolerance for stateful service, persistence and recovery mechanism is typically required, which is not fully considered in our work.

3.4 Fault Reporting

Though ultimate goal of pervasive computing is transparent (minimal user awareness) computing to the user, we believe that today’s design and implementation technology is not mature enough so that human intervening is inevitable. We defined an XML rule for fault report. Fault condition is classified according to its severity. Three severity levels are defined, Emergency, Notification, and Log. Both Emergency and Notification send notification message to users or/and operators, and Log does not. The difference between Emergency and Notification is some remedy measure should be defined for handling the situation, while Notification just notifies. Remedy measure examples include turning on a fire extinguisher and calling 911 emergency systems. Log is for a least significant fault that is just required to be logged so that the system administrator can examine it for diagnostic, error tracking, and bundle reconfiguration. Reporting methods are defined as URL service at the moment, prepared voip call based on (Session Initiation Protocol, sip), pager, and speaker sound. Figure 7 shows an example instance for XML.

```
<fault>
  <faultServerity>Emergency</faultServerity>
  <faultHandling>
    <serviceCall>
      <service>org.foo.service.ExtinguisherService</service>
      <call>start</call>
    </serviceCall>
  </faultHandling>
  <FaultNotification>
    <URL>http://smarthome.org/faultreport?$FAULT &$LOCATION</URL>
    <URL>sip:operator@smarthome.org</URL>
  </FaultNotification>
</fault>
```

Figure 7. A fault report XML instance

4. IMPLEMENTATION EXPERIENCE

We have been constructing a test bed for our smart home gateway system. The current test scenarios is limited to home automation application and the miniature house is installed sensors for gas detection, intrusion detection, and health monitoring of people (Figures 8 and 9). We extended the Oscar implementation of OSGi specification 3.0, and fully implemented our proxy based OSGi framework on embedded linux platform. The hardware and software of the implementation are summarized in Figure 10 and Table 2.
Table 2. Table captions should be placed above the table

<table>
<thead>
<tr>
<th>Item</th>
<th>Function</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>ARM-7 Core</td>
<td>EP7312, 74MHz</td>
</tr>
<tr>
<td></td>
<td>MCU</td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>Linux 2.4.22</td>
<td>Arm/bluez patch</td>
</tr>
<tr>
<td>JVM</td>
<td>Sun J2ME FP/</td>
<td>j2me_cdcfp-1_0_1-fcs</td>
</tr>
<tr>
<td></td>
<td>Kaffe</td>
<td>(May_2002)/kaffe-1.1.4</td>
</tr>
<tr>
<td>OSGi Platform</td>
<td>OSCAR 1.0</td>
<td>extensions described in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the previous chapters</td>
</tr>
<tr>
<td>Connectivity</td>
<td>Ethernet</td>
<td></td>
</tr>
<tr>
<td></td>
<td>802.11b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bluetooth</td>
<td></td>
</tr>
<tr>
<td>Local Network</td>
<td>UpnP</td>
<td>Cyberspace[9]</td>
</tr>
<tr>
<td></td>
<td>Bluetooth</td>
<td>Bluez 1.4[10]</td>
</tr>
</tbody>
</table>

5. CONCLUSION
We proposed a proxy-based approach to provide reliability support to OSGi framework. The proxy wrapper enables the framework to monitor, detect fault, and isolate failed service from other service, and provide advanced recovery. We implemented our concept into Oscar implementation and applied to smart home automation and security application. The proxy-based approach can solve system reliability issues due to the service dynamics very efficiently and thoroughly. In the near future, we will focus two areas. Firstly, we will build more application scenarios for verifying the usefulness and improving our implementation. Secondly, we will look into service state management and system resource abuse control.

6. ACKNOWLEDGMENTS
The Research was partially funded by SNUT Industrial-Academic Cooperation Program and BNSoft corp.

7. REFERENCES
[13] Web Services Coordination (WS-Coordination), Microsoft, IBM, and BEA Systems October 2004