Editor: Sumi Helal ■ University of Florida ■ helal@cise.ufl.edu

Enabling Smart Spaces with OSGi

Choonhwa Lee, David Nordstedt, and Sumi Helal

Today's pervasive computing spaces are developed primarily with proprietary technology and seem to lack a long-term vision of evolution and interoperation. The future pervasive computing environment will comprise a wide variety of devices and services from different manufacturers and developers. We must therefore achieve platform and vendor independence as well as architecture openness before pervasive computing spaces become common places.

The Open Services Gateway Initiative attempts to meet these requirements by providing a managed, extensible framework to connect various devices in a local network such as in a home, office, or automobile. By defining a standard execution environment and service interfaces, OSGi promotes the dynamic discovery and collaboration of devices and services from different sources. Moreover, the framework is designed to ensure smooth space evolution over time and to support connectivity to the outside world, allowing remote control, diagnosis, and management.

Originated in 1999 to deliver WAN services to home environments, OSGi today offers a unique opportunity to pervasive computing as a potential framework for achieving interoperability and extensibility. Through two major enhancements over the last few years, the specification now includes numerous new services and features that support various usage models. Here, we present the OSGi technology and examine several OSGi development toolkits and products. We also share our experience in using OSGi to build an open, assistive environment that supports independent living for elders.

THE FRAMEWORK

The OSGi framework provides general-purpose, secure support for deploying extensible and downloadable Java-based service applications known as *bundles*. An OSGi service platform is an instantiation of a Java virtual machine, an OSGi framework, and a set of bundles (see www.osgi.org/resources/spec_download.asp).

Running on top of a Java virtual machine, the framework provides a shared execution environment that installs, updates, and uninstalls bundles without needing to restart the system. Bundles can collaborate by providing other bundles with application components called *services*. An installed bundle might register zero or more services with the framework's service registry. This registration advertises the services and makes them discoverable through the registry so that other bundles can use them. The framework also manages dependencies among bundles and services to facilitate coordination among them.

The framework provides application developers with a consistent programming model by defining only interfaces between the framework and the services. The real implementation is left to the bundle developers; service selection and interaction decisions are made dynamically at service discovery and use time. This separation of service definition and implementation ensures that services from different service providers interoperate on the managed framework. Figure 1 depicts the OSGi service gateway placed between a WAN and LAN.

We can deploy a bundle in an OSGi service platform to provide application functions to other bundles or users. The bundle is a Java Archive (JAR) file that contains Java class files to implement zero or more services, a manifest file, and resources such as HTML pages, help files, icons, and so forth. The manifest file describes the JAR file itself and provides additional information about the bundle. Some examples of manifest headers include `Import-Package`, `Export-Package`, and `Bundle-Activator`. `Import-Package` and `Export-Package` guide the framework to resolve shared package dependencies by listing package names that must be imported and can be exported. Packages are an indivisible unit constituting an OSGi bundle. The `Bundle-Activator` header specifies a class whose `start` and `stop` methods are called by the framework to start and stop the bundle.

A bundle can register services with the framework service registry. In this case, the service implementation (that is, the service object), which is represented by its Java service interface, is what actually gets registered. Bundles can discover services offered by each other by querying the service registry using a simple service discovery interface. When a bun-

1536-1268/03/$17.00 © 2003 IEEE ■ *Published by the IEEE CS and IEEE ComSoc*

STANDARDS, TOOLS & BEST PRACTICES

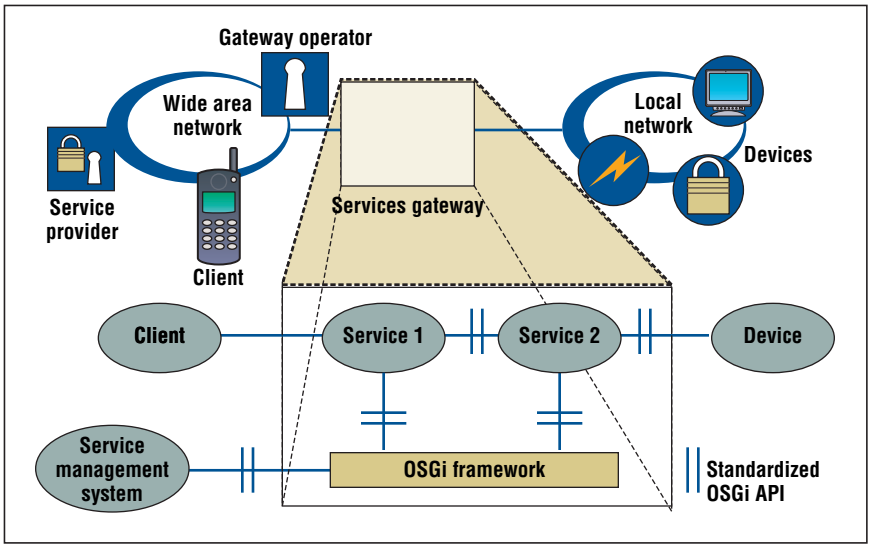


Figure 1. The OSGi framework and services.

dle queries the registry, it obtains references to actual service objects registered under the desired service interface name. Besides the interface name, we can further describe services using a collection of key-value pairs. We can then match the services' properties against a filter parameter to narrow down the query results. The OSGi filter is based on the LDAP search filter's string representation.

The framework manages dependency among bundles that offer and use a given service. For example, when a bundle is stopped, the framework automatically unregisters all services that the bundle registered. Also, service events can notify a bundle when a service from other bundles is registered, modified, or unregistered. Figure 2 shows how the service registry is used to advertise and discover services.[1]

The simplified code fragments in Figure 3 are an implementation of a sample "Hello World" service along with the bundles that provide and use it. Figures 3a and 3b show the service interface definition and the service implementation. The bundle in Figure 3c registers the service implementation object with the framework service registry along with a service property. The bundle in Figure 3d shows how to discover the service object using the service interface name and service property. As shown, every bundle should implement the BundleActivator interface, which defines the start and stop methods

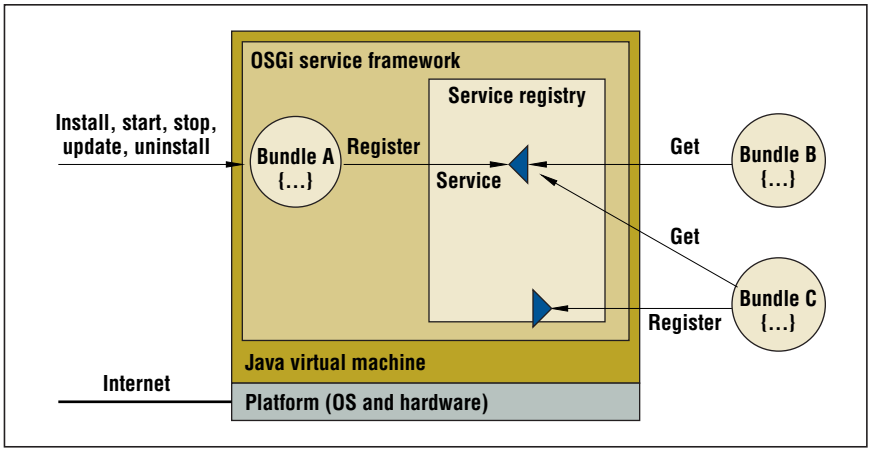


Figure 2. Bundle collaboration through service registry.

SPECIFICATIONS

Since the OSGi specifications' first release in May 2000, they have gone through two major updates, in October 2001 and March 2003. The latest specs provide several reference architectures, including a large service delivery network managed by an operator and a home or office network to integrate various computing devices. Although OSGi can apply to various scenarios, perhaps the most appealing use model is the remote-management reference architecture. It lets an operator manage a large network of service platforms and services that different service providers supply. *Remote managers* are at a central site, and they communicate with *manager agents* on remote target service platforms. A manager agent is a set of bundles that provides remote management of the service platform to its central remote manager. The remote management features, OSGi 3.0's most conspicuous development, appear in the specification documents *Configuration Admin Service*, *Wire Admin Service*, *Start Level Service*, *Initial Provisioning*, *Preference Service*, *Package Admin Service*, and *Permission Admin Service*. Another noteworthy feature, driven by the automotive industry, is the *Position* specification to indicate a vehicle's position and movement.

Table 1 summarizes the services and classes that the latest OSGi specification (release 3.0) defines, all of which are available to bundle developers.

DEVELOPMENT TOOLKITS

The OSGi Web site (www.osgi.org) lists OSGi service platform developer kits available from several vendors. We look at a few from Prosyst, IBM, and Sun Microsystems.

ProSyst mBedded Builder

Prosyst has a complete line of prod-

ucts for an OSGi 2.0 system, which includes their mBedded Server, mBedded Remote Manager, and mBedded Builder. The mBedded Server runs on many operating systems and provides the OSGi service platform. The mBedded Remote Manager provides for easy management of a possibly large network of OSGi service platforms. The mBedded Builder is a complete, integrated development environment with many extensions for application developers to build services for an OSGi system. In addition to a complete GUI IDE for developing Java applications, there is support for CVS (Concurrent Versions System), custom device GUI design, and creating OSGi service bundles.

Java Embedded Server

Sun Microsystems produced its Java Embedded Server before the OSGi consortium was founded. In fact, Sun was a founding OSGi member, and JES significantly influenced the OSGi specification. JES has evolved to the current 2.0 version, which is fully compliant with OSGi 1.0. The current version is a free download at http://java.sun.com/jes and is supported on the Solaris 2.7 and Windows NT platforms. There is a free plug-in to the SunONE Studio IDE, also available on Sun's Web site.

IBM SMF Bundle Developer

IBM's Websphere Device Developer helps create mobile applications for many devices, including the J2ME applications that have become so popular in the past couple of years. This developer product now also supports IBM's own OSGi implementation, Service Management Framework, through the SMF Bundle Developer plug-in. This plug-in allows GUI creation and manipulation of OSGi bundles, manifest headers, and testing. The IBM SMF implements OSGi 3.0 but omits many features, such as Start Level, URL Handlers Service, I/O Connector Service, Wire Admin Service, Namespace, Jini Driver Service, UPnP Device Service, and Initial Provisioning support.

```
package srvs;
public interface HelloWorldService {
  public void foo();
}
(a)

public HelloWorldServiceImpl implement HelloWorldService {
public void foo() {
  System.out.println("Hello World");
  }
}
(b)

public class HelloWorldProvider implements BundleActivator {
  private ServiceRegistration reg = null;

  public void start(BundleContext ctxt) throws Exception {
    HelloWorldService fooSrv = new HelloWorldServiceImpl();

    Properties props = new Properties();
    Props.put("description", "sample");

    reg = ctxt.registerService("srvs.HelloWorldService", fooSrv, props);
  }

  public void stop(BundleContext ctxt) throws Exception {
    if (reg != null)
      reg.unregister();
  }
}
(c)

public class HelloWorldServiceUser implements BundleActivator {
  public void start(BundleContext ctxt) throws Exception {
    ServiceReference[] ref = ctxt.getServiceReference("srvs.HelloWorldService", "(description=sample)");
    HelloWorldService fooSrv = (HelloWorldService) ctxt.getService(ref[0]);
    fooSrv.foo();
  }

  public void stop(BundleContext ctxt) throws Exception {
  }
}
(d)
```

Figure 3. A sample "Hello World" OSGi service: (a) interface; (b) implementation; (c) provider bundle; and (d) user bundle.

ADOPTION AND PRODUCTS

Numerous member companies developing and supplying OSGi-based products back the OSGi consortium.

The E2-Home Project in Stockholm executed the first commercial deployment of a consumer-oriented OSGi service platform using the Gatespace e-Services embedded software (www.gatespace.com). This project, which

STANDARDS, TOOLS & BEST PRACTICES

TABLE 1
OSGi Specifications.

Specification (first release)	Description
Package Admin Service (2)	Provides an interface to resolve dependencies between bundles exporting and importing packages. For instance, it refreshes packages exported by a bundle that is being unregistered or updated.
Start Level Service (3)	Controls starting or stopping bundle sequences. It assigns a bundle to a *bundle start level* and the framework to an *active start level*. It starts bundles that have a start level equal to or less than the active start level.
Permission Admin Service (2)	A bundle is assigned a set of permissions that decide whether the bundle is authorized to execute privileged code. Permission Admin Service provides an interface to manipulate the framework's repository of per-bundle permissions.
URL Handlers Service (3)	Registers service objects that support new URL schemes and corresponding content-typed stream handlers in an OSGi service platform.
Log Service (1)	Provides two services—Log Service to record log information and Log Reader Service to retrieve the information.
Configuration Admin Service (2)	Configuration data is set of properties that a remote agent or other applications in an OSGi environment maintain. A Configuration Admin Service instance hands over the configuration data to bundles on their registration or when their configuration changes at a later time.
Device Access (1)	Specifies a device model based on the device manager, which controls automatic attachment of a suitable device driver service to a newly registered device service.
User Admin Service (2)	Defines a flexible authentication to adopt different authentication schemes. Once authenticated, a bundle uses its role-based authorization to verify if the user is authorized to perform the requested action.
I/O Connector Service (3)	Defines a flexible, extendable communication API based on the J2ME Connector framework of the javax.microedition.io package.
HTTP Service (1)	Provides APIs for bundles to register servlets or resources such as static HTML pages, images, sounds, and so on, so that a standard Web browser can access them in an OSGi service platform.
Preference Service (2)	Provides a bundle with persistent storage of named data values. Unlike the java.util.Properties class, it supports a hierarchical naming model, and its key/value pairs can be stored in a remote machine.
Wire Admin Service (3)	Intended for user interfaces or management applications, Wire Admin Service controls the wiring of services—for example, wiring data-producing services to data-consuming services. It enables dynamically configurable collaboration among bundles.
XML Parser Service (3)	Defines how XML, SAX, and DOM parsers can be provided and used in an OSGi service platform.
Service Tracker (2)	A utility service that tracks the registration, unregistration, and modification of services of interest. Given a set of services, a ServiceTracker object begins the tracking by listening to ServiceEvents from the framework, and the actions in the event of service changes can be customized.
Measurement and State (3)	Allows a consistent handling and exchange of a wide variety of measurements. Any measurement can be represented by the seven basic SI units and derived units. A State object holds integer values to represent discrete states.
Position (3)	Handles geographical positions and movements in OSGi applications. Based on WGS-84 GPS code, a Position object contains latitude, longitude, altitude, track, and speed fields.
Jini Driver Service (3)	Defines a bridge between Jini and OSGi. More specifically, it defines an interface for Jini-to-OSGi service import and OSGi-to-Jini service export.
UPnP Device Service (3)	Understands UPnP protocol to transform UPnP services to OSGi services and vice versa.

launched in 2001, is used in 180 condominiums for services such as energy management, home automation, and community services. Members of the housing project can access the Web to monitor their utility consumption, reserve common areas such as the laundry and sauna, and order store items for delivery. All these services, including burglar alarm and email access, are available from a convenient touchscreen computer in the kitchen.

Cisco has deployed OSGi service platform technology in their CiscoWorks 2000 Service Management Solution, which manages service agreement levels between enterprise and other networks to ensure quality of service. Cisco is also using the Gatespace software in their system.

Whirlpool (www.whirlpoolcorp.com) uses IBM's OSGi service platform for its Home Solutions product line. Featured services will include Internet-enabled appliance controls, Internet connection sharing, and home firewall capabilities. Whirlpool plans to ship the system later this year.

BSH (Bosch und Siemens Hausgeräte) in Europe has presented a full line of net-

worked home appliances, code-named Smart@Home. The appliances will be available by the end of 2003, according to BSH. The OSGi software for the gateway, developed by BSH in conjunction with ProSyst and Siemens, lets users access their appliances with a WebPad inside the house or a WAP-capable cell phone from outside.

InterComponentWare uses the Prosyst OSGi technology in their LifeSensor product (www.lifesensor.com), which transfers medical information from patients' medical devices to remote caregivers.

OSGI VERSUS JINI

Based on the Java platform, both OSGi and Jini tackle the same service discovery problem. Services represented by Java interfaces are advertised and discovered through a service registry according to both technologies.

Despite this seeming resemblance, OSGi targets more closed and static environments than Jini does. The OSGi framework provides a managed execution environment for service discovery and collaboration. To discover each other, bundles that offer and use a service should be installed and started in the same framework (that is, in the same Java virtual machine), usually by a remote operator or owner. Therefore, the discovery scope is within a Java virtual machine. Of course, a bundle might be a proxy to represent external devices or services to the OSGi world. In the case of Jini, clients and services live in different Java virtual machines. The physical span of a Jini community is typically determined by the administratively scoped IP multicast range. In other words, OSGi framework provides an execution environment as well as a rendezvous point for services, while a Jini service registry serves only as a rendezvous point between services and clients.

CASE STUDY: MATILDA'S SMART HOUSE

Matilda's Smart House (see Figure 4) is a multidisciplinary project led by Sumi Helal and William Mann at the University of Florida (see www.rerc.ufl.edu). The project explores the use of emerging smart phones and other wireless technologies to create "magic wands" that let elder people with disabilities interact with, monitor, and control their surroundings. By integrating the smart phone with smart environments, elders can turn appliances on and off, check and change the status of locks on doors and windows, and order groceries. The project also tests smart phones that can remind users to take medications or call in prescription drug refills automatically. Matilda is an "elder" robot with an onboard computer and a vest fitted with location sensors. She is being used as a research instrument to experiment with indoor location tracking research.

The house is instrumented with various sensors and devices, including J2ME smart phones as user devices, ultrasonic location sensors, X-10 controlled devices (door, mailbox, curtain, lamp, and radio), and networked devices (microwave, fridge, LCD wall-mounted displays, and cameras). The OSGi framework was a nearly ideal match for our need for an extensible software infrastructure that can adapt to environmental changes (such as introducing and integrating new devices and services). It also supports remote monitoring and administration by fam-

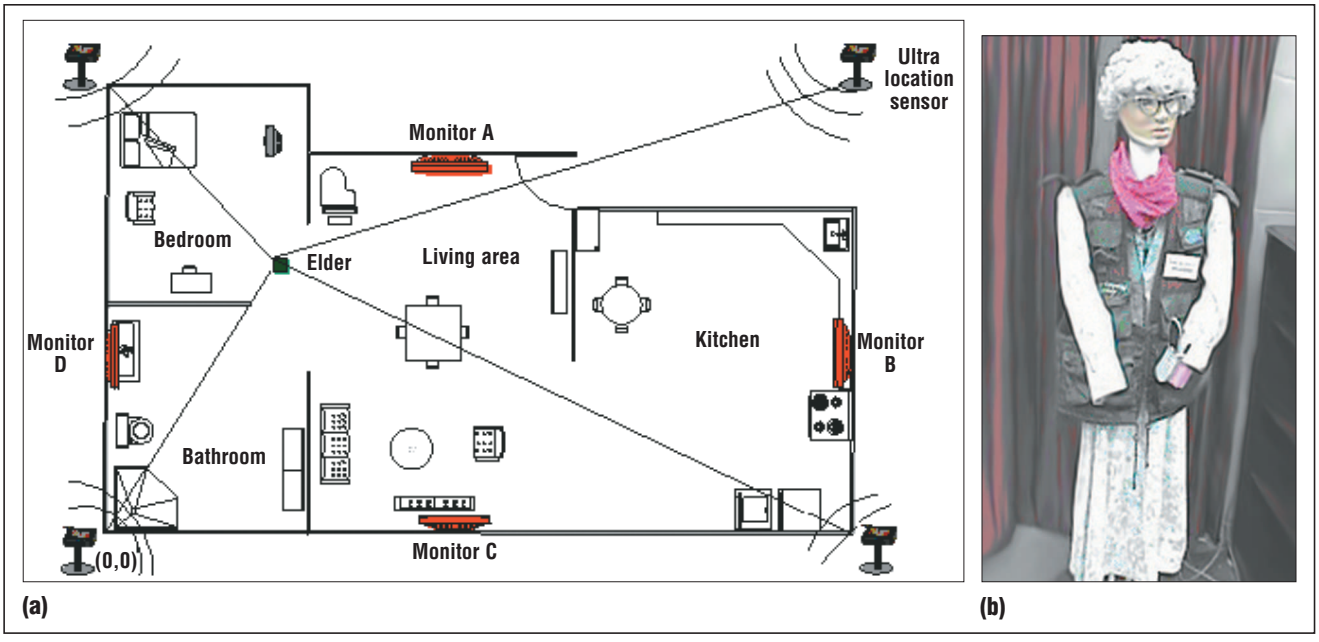


Figure 4. (a) Matilda's Smart House and (b) Matilda.

STANDARDS, TOOLS & BEST PRACTICES

ily members and caregivers.

The smart space built on the OSGi framework provided a solid infrastructure so that the project team could focus more on integrating the smart phone (and by extension, the resident) with the smart environments and various pervasive computing applications.[2] For example, the LCD display that Matilda is facing provides a medication reminder, with the smart house sensing her orientation and location. The house issues an audio warning if she picks up the wrong medicine bottle, which is detected by an RFID reader attached to her phone. Also, Matilda can use her phone to open the door upon delivery of her automatically requested prescription drug refills. For most of these applications, we coded them after an effortless and simple composition of bundle services.

At the University of Florida, we built Matilda's Smart House prototype using Sun Microsystems' JES (see wwws.sun.com/software/embeddedserver), which is a reference implementation of the OSGi Specification Release 1.0. However, the specification was missing a generic interbundle eventing mechanism. The OSGi specification supports only basic events including ServiceEvent, BundleEvent, and FrameworkEvent. A ServiceEvent notifies when a service is registered, modified, or unregistered. A BundleEvent indicates a bundle's state transition among Installed, Uninstalled, Resolved, Starting, Stopping, or ActiveState. A FrameworkEvent reports some changes in the framework.

Early in the house's development, we discovered the need for an extensible event mechanism that chains services together according to their application-specific data producer and consumer relationship. Data that a source service produces flows through an event chain toward a sink service. We modeled this interservice communication as a generic EventBroker service to mediate the producer and consumer communication.

One example of event sources is a location service. When Matilda's position changes, the location service updates the event broker on her new position, which in turn notifies potential client applications subscribing to the location event. Filters that the subscribers provide notify client applications of only events of interest. For example, bathroom and hygiene applications will want to be notified only when Matilda is in any of the bathroom areas. The Wire Admin Service in OSGi service platform 3.0 now addresses the need for an effective interservice eventing mechanism, which facilitates service composition. A Wire object connects a Producer and Consumer service. It also supports advanced features such as filter-based wire flow control and data type converters.

OSGi technology has gained significant momentum and is ready for wide market deployment. The latest specification is receiving wider acceptance and attention.

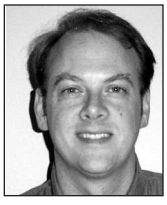
We can enrich services available in an OSGi service platform by importing other services from other service discovery networks. Newly introduced in Release 3.0, the Jini Driver Service is a bridge between the OSGi and the Jini world. Similarly, introduced in the same release, the UPnP Device Service bridges the OSGi and UPnP worlds. Because Bluetooth is yet another important technology in home environments, we are looking at an OSGi-to-Bluetooth gateway. In addition, we are planning to upgrade our current smart space implementation to use more versatile services of the latest OSGi specification, including the Wire Admin Service.

OSGi technology has gained significant momentum and is ready for wide market deployment. Driven by recent advances in wireless and mobile technology, as well as clearer business models and market needs, the latest specification is receiving wider acceptance and attention. Several companies have developed their OSGi-based products and solutions already, and many are exploring or toying with it. Candidate pervasive computing applications include home automation and health care, remote service delivery and administration, and automotive networks. The bottom line: OSGi presents a perfect infrastructure to integrate various devices and sensors and to cope with the dynamism inherent in future pervasive computing environments. P

REFERENCES

1. D. Marples and P. Kriens, "The Open Services Gateway Initiative: An Introductory Review," *IEEE Communications Magazine*, vol. 39, no. 12, Dec. 2001, pp. 110–114.
2. S. Helal et al., "Enabling Location-Aware Pervasive Computing Applications for the Elderly," *Proc. IEEE Int'l Conf. Pervasive Computing and Communications* (PerCom 2003), IEEE CS Press, 2003, pp. 531–538.

Choonhwa Lee is a post doctoral research fellow at the University of Florida. Contact him at chl@cise.ufl.edu.



David Nordstedt is a chief software architect at Phoneomena. Contact him at david@phoneomena.com.

Sumi Helal is an associate professor in the Computer and Information Science and Engineering Department at the University of Florida. He is also the founder, president, and CEO of Phoneomena. Contact him at helal@cise.ufl.edu; www.cise.ufl.edu/~helal.